

Laravel 10.x

Offline Documentation

Last updated on: Wed Nov 12 2025

Created & maintained by — [Mohammad Nurul Islam \(Shihan\)](#).

Preface

Why PDF version?

As a full stack web (and backend service) developer, I often need to create api & admin interface for small to medium web applications in shortest possible time. And due the expressive syntax & the rapid development tools provided, I find [Laravel](#) to be very useful for this purpose.

Also for myself, I sometimes intentionally turn off internet while writing code to focus on work to meet deadline. During that isolated time, I need offline versions of documentation of various tools, libraries and framework. Even though, for most of the other tools, libraries & frameworks offline documentation is available, for Laravel it was never true. To solve this issue, few years back I created & maintained offline downloadable documentation of all Laravel versions available then and published via [Leanpub](#) as free (with 0.0 minimum price) for others, with scripted processing on the markdown files from the [Laravel documentation repo](#) available in github. But, unfortunately with some successive policy changes by Leanpub authority for the authors, it was becoming impossible for me to maintain already published versions or create new books for new versions of Laravel. Also, due to professional requirement, I had to switch to various other technologies and didn't had to work with Laravel for a few years. So, I eventually give up maintaining those Leanpub publications of Offline Documentation of Laravel.

Recently I got back to Laravel for some new projects and missed the Offline Documentation again, while getting myself up to date with new features available in new versions of Laravel. So, this time, instead of depending on third party publishing platforms, I developed a new tool myself for automating the required processing and creation of PDFs from various branches of the [documentation repo](#). This file is result of one such conversion.



In [laravel-docs-in-pdf](#) repo, you can always find up-to-date PDF files corresponding to documentation for each version of Laravel and download copies for offline use. If you find these PDF documentation files useful and want to show some support, please give [the repo](#) a star and share with your social/professional network.

You can also [watch the repo](#) and/or [follow](#) me on [Github](#), [Twitter](#) or [Linkedin](#), to get [notification](#) about future releases of the PDF version of Laravel documentation.

Any kind of **appreciation** from the community will **encourage & motivate me** to continue maintaining this project **regularly**. To show your generous support, you can of course



[Buy me a coffee](#)

License & copyright

All the PDF files available in [laravel-docs-in-pdf](#) repo are licensed under [The Unlicense](#) license and available in [public domain](#). Feel free to copy, modify, distribute or whatever you want to do with these files.

But, please be aware that, the documentation content of all the PDF files (beside the cover & preface pages) are not covered with this license and is licensed under the original license of [Laravel documentation](#) as defined in the [original documentation's repo](#).

Table of Contents

- [Prologue](#)
 - [Release Notes](#)
 - [Upgrade Guide](#)
 - [Contribution Guide](#)
- [Getting Started](#)
 - [Installation](#)
 - [Configuration](#)
 - [Directory Structure](#)
 - [Frontend](#)
 - [Starter Kits](#)
 - [Deployment](#)
- [Architecture Concepts](#)
 - [Request Lifecycle](#)
 - [Service Container](#)
 - [Service Providers](#)
 - [Facades](#)
- [The Basics](#)
 - [Routing](#)
 - [Middleware](#)
 - [CSRF Protection](#)
 - [Controllers](#)
 - [Requests](#)
 - [Responses](#)
 - [Views](#)
 - [Blade Templates](#)
 - [Asset Bundling](#)
 - [URL Generation](#)
 - [Session](#)
 - [Validation](#)
 - [Error Handling](#)
 - [Logging](#)
- [Digging Deeper](#)
 - [Artisan Console](#)
 - [Broadcasting](#)
 - [Cache](#)
 - [Collections](#)
 - [Contracts](#)
 - [Events](#)
 - [File Storage](#)
 - [Helpers](#)
 - [HTTP Client](#)
 - [Localization](#)
 - [Mail](#)
 - [Notifications](#)
 - [Package Development](#)
 - [Processes](#)
 - [Queues](#)
 - [Rate Limiting](#)
 - [Strings](#)

- [Task Scheduling](#)
- [Security](#)
 - [Authentication](#)
 - [Authorization](#)
 - [Email Verification](#)
 - [Encryption](#)
 - [Hashing](#)
 - [Password Reset](#)
- [Database](#)
 - [Getting Started](#)
 - [Query Builder](#)
 - [Pagination](#)
 - [Migrations](#)
 - [Seeding](#)
 - [Redis](#)
- [Eloquent ORM](#)
 - [Getting Started](#)
 - [Relationships](#)
 - [Collections](#)
 - [Mutators / Casts](#)
 - [API Resources](#)
 - [Serialization](#)
 - [Factories](#)
- [Testing](#)
 - [Getting Started](#)
 - [HTTP Tests](#)
 - [Console Tests](#)
 - [Browser Tests](#)
 - [Database](#)
 - [Mocking](#)
- [Packages](#)
 - [Breeze](#)
 - [Cashier \(Stripe\)](#)
 - [Cashier \(Paddle\)](#)
 - [Dusk](#)
 - [Envoy](#)
 - [Fortify](#)
 - [Folio](#)
 - [Homestead](#)
 - [Horizon](#)
 - [Jetstream](#)
 - [MCP](#)
 - [Mix](#)
 - [Octane](#)
 - [Passport](#)
 - [Pennant](#)
 - [Pint](#)
 - [Precognition](#)
 - [Prompts](#)
 - [Pulse](#)
 - [Reverb](#)
 - [Sail](#)
 - [Sanctum](#)

- [Scout](#)
- [Socialite](#)
- [Telescope](#)
- [Valet](#)
- [API Documentation](#)

Chapter 1

Prologue

Release Notes

- [Versioning Scheme](#)
- [Support Policy](#)
- [Laravel 10](#)

Versioning Scheme

Laravel and its other first-party packages follow [Semantic Versioning](#). Major framework releases are released every year (~Q1), while minor and patch releases may be released as often as every week. Minor and patch releases should **never** contain breaking changes.

When referencing the Laravel framework or its components from your application or package, you should always use a version constraint such as `^10.0`, since major releases of Laravel do include breaking changes. However, we strive to always ensure you may update to a new major release in one day or less.

Named Arguments

[Named arguments](#) are not covered by Laravel's backwards compatibility guidelines. We may choose to rename function arguments when necessary in order to improve the Laravel codebase. Therefore, using named arguments when calling Laravel methods should be done cautiously and with the understanding that the parameter names may change in the future.

Support Policy

For all Laravel releases, bug fixes are provided for 18 months and security fixes are provided for 2 years. For all additional libraries, including Lumen, only the latest major release receives bug fixes. In addition, please review the database versions [supported by Laravel](#).

Version	PHP (*)	Release	Bug Fixes Until	Security Fixes Until
8	7.3 - 8.1	September 8th, 2020	July 26th, 2022	January 24th, 2023
9	8.0 - 8.2	February 8th, 2022	August 8th, 2023	February 6th, 2024
10	8.1 - 8.3	February 14th, 2023	August 6th, 2024	February 4th, 2025
11	8.2 - 8.4	March 12th, 2024	September 3rd, 2025	March 12th, 2026

End of life

Security fixes only

(*) Supported PHP versions

Laravel 10

As you may know, Laravel transitioned to yearly releases with the release of Laravel 8. Previously, major versions were released every 6 months. This transition is intended to ease the maintenance burden on the community and challenge our development team to ship amazing, powerful new features without introducing breaking changes. Therefore, we have shipped a variety of robust features to Laravel 9 without breaking backwards compatibility.

Therefore, this commitment to ship great new features during the current release will likely lead to future "major" releases being primarily used for "maintenance" tasks such as upgrading upstream dependencies, which can be seen in these release notes.

Laravel 10 continues the improvements made in Laravel 9.x by introducing argument and return types to all application skeleton methods, as well as all stub files used to generate classes throughout the framework. In addition, a new,

developer-friendly abstraction layer has been introduced for starting and interacting with external processes. Further, Laravel Pennant has been introduced to provide a wonderful approach to managing your application's "feature flags".

PHP 8.1

Laravel 10.x requires a minimum PHP version of 8.1.

Types

Application skeleton and stub type-hints were contributed by [Nuno Maduro](#).

On its initial release, Laravel utilized all of the type-hinting features available in PHP at the time. However, many new features have been added to PHP in the subsequent years, including additional primitive type-hints, return types, and union types.

Laravel 10.x thoroughly updates the application skeleton and all stubs utilized by the framework to introduce argument and return types to all method signatures. In addition, extraneous "doc block" type-hint information has been deleted.

This change is entirely backwards compatible with existing applications. Therefore, existing applications that do not have these type-hints will continue to function normally.

Laravel Pennant

Laravel Pennant was developed by [Tim MacDonald](#).

A new first-party package, Laravel Pennant, has been released. Laravel Pennant offers a light-weight, streamlined approach to managing your application's feature flags. Out of the box, Pennant includes an in-memory `array` driver and a `database` driver for persistent feature storage.

Features can be easily defined via the `Feature::define` method:

```
use Laravel\Pennant\Feature;
use Illuminate\Support\Lottery;

Feature::define('new-onboarding-flow', function () {
    return Lottery::odds(1, 10);
});
```

Once a feature has been defined, you may easily determine if the current user has access to the given feature:

```
if (Feature::active('new-onboarding-flow')) {
    // ...
}
```

Of course, for convenience, Blade directives are also available:

```
@feature('new-onboarding-flow')
<div>
    <!-- ... -->
</div>
@endfeature
```

Pennant offers a variety of more advanced features and APIs. For more information, please consult the [comprehensive Pennant documentation](#).

Process Interaction

The process abstraction layer was contributed by [Nuno Maduro](#) and [Taylor Otwell](#).

Laravel 10.x introduces a beautiful abstraction layer for starting and interacting with external processes via a new `Process` facade:

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

return $result->output();
```

Processes may even be started in pools, allowing for the convenient execution and management of concurrent processes:

```
use Illuminate\Process\Pool;
use Illuminate\Support\Facades\Process;

[$first, $second, $third] = Process::concurrently(function (Pool $pool) {
    $pool->command('cat first.txt');
    $pool->command('cat second.txt');
    $pool->command('cat third.txt');
});

return $first->output();
```

In addition, processes may be faked for convenient testing:

```
Process::fake();

// ...

Process::assertRan('ls -la');
```

For more information on interacting with processes, please consult the [comprehensive process documentation](#).

Test Profiling

Test profiling was contributed by [Nuno Maduro](#).

The Artisan `test` command has received a new `--profile` option that allows you to easily identify the slowest tests in your application:

```
php artisan test --profile
```

For convenience, the slowest tests will be displayed directly within the CLI output:

```

nunomaduro@nunomaduro:~/Work/code/laravel

PASS Tests\Feature\UpdateTeamNameTest
✓ team names can be updated 0.02s

Tests: 1 skipped, 43 passed (78 assertions)
Duration: 10.02s

Top 10 slowest tests:
Tests\Unit\ExampleTest > that true is true 5.01s
Tests\Feature\BrowserSessionsTest > other browser sessions can be logged 3.04s
Tests\Feature\ApiTokenPermissionsTest > api token permissions can be upd 0.23s
Tests\Feature\PasswordConfirmationTest > password is not confirmed with 0.22s
Tests\Feature\AuthenticationTest > users can authenticate using the logi 0.13s
Tests\Feature\UpdatePasswordTest > new passwords must match 0.13s
Tests\Feature\UpdatePasswordTest > current password must be correct 0.13s
Tests\Feature\DeleteAccountTest > user accounts can be deleted 0.08s
Tests\Feature\UpdatePasswordTest > password can be updated 0.08s
Tests\Feature\DeleteAccountTest > correct password must be provided befo 0.07s

(91.10% of 10.02s) 9.12s

```

Pest Scaffolding

New Laravel projects may now be created with Pest test scaffolding by default. To opt-in to this feature, provide the `--pest` flag when creating a new application via the Laravel installer:

```
laravel new example-application --pest
```

Generator CLI Prompts

Generator CLI prompts were contributed by [Jess Archer](#).

To improve the framework's developer experience, all of Laravel's built-in `make` commands no longer require any input. If the commands are invoked without input, you will be prompted for the required arguments:

```
php artisan make:controller
```

Horizon / Telescope Facelift

[Horizon](#) and [Telescope](#) have been updated with a fresh, modern look including improved typography, spacing, and design:

The screenshot shows the Laravel Horizon dashboard for a Laravel application. The top navigation bar indicates the URL as "Not Secure — laravel.test". The main header is "Laravel Horizon - Laravel". On the left, a sidebar menu lists several options: Dashboard (selected), Monitoring, Metrics, Batches, Pending Jobs, Completed Jobs, Silenced Jobs, and Failed Jobs. The "Overview" section displays real-time metrics: Jobs Per Minute (246), Jobs Past Hour (1,100), Failed Jobs Past 7 Days (0), Status (Active), Total Processes (2), Max Wait Time (-), Max Runtime (default), and Max Throughput (default). Below this is the "Current Workload" section, which shows a single queue named "default" with 200 jobs, 2 processes, and a wait time of "A few seconds". At the bottom, there is a table for the supervisor named "nunomaduro/home-xaZQ", which has one entry: supervisor-1, running on the default queue with 2 processes and Auto balancing.

Supervisor	Queues	Processes	Balancing
supervisor-1	default	2	Auto

Upgrade Guide

- [Upgrading to 10.0 from 9.x](#)

High Impact Changes

- [Updating Dependencies](#)
- [Updating Minimum Stability](#)

Medium Impact Changes

- [Database Expressions](#)
- [Model "Dates" Property](#)
- [Monolog 3](#)
- [Redis Cache Tags](#)
- [Service Mocking](#)
- [The Language Directory](#)

Low Impact Changes

- [Closure Validation Rule Messages](#)
- [Form Request `after` Method](#)
- [Public Path Binding](#)
- [Query Exception Constructor](#)
- [Rate Limiter Return Values](#)
- [The `Redirect::home` Method](#)
- [The `Bus::dispatchNow` Method](#)
- [The `registerPolicies` Method](#)
- [UUID Columns](#)

Upgrading to 10.0 from 9.x

Estimated Upgrade Time: 10 Minutes



We attempt to document every possible breaking change. Since some of these breaking changes are in obscure parts of the framework only a portion of these changes may actually affect your application. Want to save time? You can use [Laravel Shift](#) to help automate your application upgrades.

Updating Dependencies

Likelihood Of Impact: High

PHP 8.1.0 Required

Laravel now requires PHP 8.1.0 or greater.

Composer 2.2.0 Required

Laravel now requires [Composer](#) 2.2.0 or greater.

Composer Dependencies

You should update the following dependencies in your application's `composer.json` file:

- `laravel/framework` to `^10.0`
- `laravel/sanctum` to `^3.2`
- `doctrine/dbal` to `^3.0`
- `spatie/laravel-ignition` to `^2.0`
- `laravel/passport` to `^11.0` ([Upgrade Guide](#))
- `laravel/ui` to `^4.0`

If you are upgrading to Sanctum 3.x from the 2.x release series, please consult the [Sanctum upgrade guide](#).

Furthermore, if you wish to use [PHPUnit 10](#), you should delete the `processUncoveredFiles` attribute from the `<coverage>` section of your application's `phpunit.xml` configuration file. Then, update the following dependencies in your application's `composer.json` file:

- `nunomaduro/collision` to `^7.0`
- `phpunit/phpunit` to `^10.0`

Finally, examine any other third-party packages consumed by your application and verify you are using the proper version for Laravel 10 support.

Minimum Stability

You should update the `minimum-stability` setting in your application's `composer.json` file to `stable`. Or, since the default value of `minimum-stability` is `stable`, you may delete this setting from your application's `composer.json` file:

```
"minimum-stability": "stable",
```

Application

Public Path Binding

Likelihood Of Impact: Low

If your application is customizing its "public path" by binding `path.public` into the container, you should instead update your code to invoke the `usePublicPath` method offered by the `Illuminate\Foundation\Application` object:

```
app()->usePublicPath(__DIR__.'/public');
```

Authorization

The `registerPolicies` Method

Likelihood Of Impact: Low

The `registerPolicies` method of the `AuthServiceProvider` is now invoked automatically by the framework. Therefore, you may remove the call to this method from the `boot` method of your application's `AuthServiceProvider`.

Cache

Redis Cache Tags

Likelihood Of Impact: Medium

Usage of `Cache::tags()` is only recommended for applications using Memcached. If you are using Redis as your application's cache driver, you should consider moving to Memcached or upgrade your application to Laravel [12.30.0](#).

Database

Database Expressions

Likelihood Of Impact: Medium

Database "expressions" (typically generated via `DB::raw`) have been rewritten in Laravel 10.x to offer additional functionality in the future. Notably, the grammar's raw string value must now be retrieved via the expression's `getValue(Grammar $grammar)` method. Casting an expression to a string using `(string)` is no longer supported.

Typically, this does not affect end-user applications; however, if your application is manually casting database expressions to strings using `(string)` or invoking the `__toString` method on the expression directly, you should update your code to invoke the `getValue` method instead:

```
use Illuminate\Support\Facades\DB;

$expression = DB::raw('select 1');

$string = $expression->getValue(DB::connection()->getQueryGrammar());
```

Query Exception Constructor

Likelihood Of Impact: Very Low

The `Illuminate\Database\QueryException` constructor now accepts a string connection name as its first argument. If your application is manually throwing this exception, you should adjust your code accordingly.

ULID Columns

Likelihood Of Impact: Low

When migrations invoke the `ulid` method without any arguments, the column will now be named `ulid`. In previous releases of Laravel, invoking this method without any arguments created a column erroneously named `uuid`:

```
$table->ulid();
```

To explicitly specify a column name when invoking the `ulid` method, you may pass the column name to the method:

```
$table->ulid('ulid');
```

Eloquent

Model "Dates" Property

Likelihood Of Impact: Medium

The Eloquent model's deprecated `$dates` property has been removed. Your application should now use the `$casts` property:

```
protected $casts = [
    'deployed_at' => 'datetime',
];
```

Localization

The Language Directory

Likelihood Of Impact: None

Though not relevant to existing applications, the Laravel application skeleton no longer contains the `lang` directory by default. Instead, when writing new Laravel applications, it may be published using the `lang:publish` Artisan command:

```
php artisan lang:publish
```

Logging

Monolog 3

Likelihood Of Impact: Medium

Laravel's Monolog dependency has been updated to Monolog 3.x. If you are directly interacting with Monolog within your application, you should review Monolog's [upgrade guide](#).

If you are using third-party logging services such as BugSnag or Rollbar, you may need to upgrade those third-party packages to a version that supports Monolog 3.x and Laravel 10.x.

Queues

The `Bus::dispatchNow` Method

Likelihood Of Impact: Low

The deprecated `Bus::dispatchNow` and `dispatch_now` methods have been removed. Instead, your application should use the `Bus::dispatchSync` and `dispatch_sync` methods, respectively.

The `dispatch()` Helper Return Value

Likelihood Of Impact: Low

Invoking `dispatch` with a class that does not implement `Illuminate\Contracts\Queue` would previously return the result of the class's `handle` method. However, this will now return an `Illuminate\Foundation\Bus\PendingBatch` instance. You may use `dispatch_sync()` to replicate the previous behavior.

Routing

Middleware Aliases

Likelihood Of Impact: Optional

In new Laravel applications, the `$routeMiddleware` property of the `App\Http\Kernel` class has been renamed to `$middlewareAliases` to better reflect its purpose. You are welcome to rename this property in your existing applications; however, it is not required.

Rate Limiter Return Values

Likelihood Of Impact: Low

When invoking the `RateLimiter::attempt` method, the value returned by the provided closure will now be returned by the method. If nothing or `null` is returned, the `attempt` method will return `true`:

```
$value = RateLimiter::attempt('key', 10, fn () => ['example'], 1);

$value; // ['example']
```

The `Redirect::home` Method

Likelihood Of Impact: Very Low

The deprecated `Redirect::home` method has been removed. Instead, your application should redirect to an explicitly named route:

```
return Redirect::route('home');
```

Testing

Service Mocking

Likelihood Of Impact: Medium

The deprecated `MocksApplicationServices` trait has been removed from the framework. This trait provided testing methods such as `expectsEvents`, `expectsJobs`, and `expectsNotifications`.

If your application uses these methods, we recommend you transition to `Event::fake`, `Bus::fake`, and `Notification::fake`, respectively. You can learn more about mocking via fakes in the corresponding documentation for the component you are attempting to fake.

Validation

Closure Validation Rule Messages

Likelihood Of Impact: Very Low

When writing closure based custom validation rules, invoking the `$fail` callback more than once will now append the messages to an array instead of overwriting the previous message. Typically, this will not affect your application.

In addition, the `$fail` callback now returns an object. If you were previously type-hinting the return type of your validation closure, this may require you to update your type-hint:

```
public function rules()
{
    'name' => [
        function ($attribute, $value, $fail) {
            $fail('validation.translation.key')->translate();
        },
    ],
}
```

Validation Messages and Closure Rules

Likelihood Of Impact: Very Low

Previously, you could assign a failure message to a different key by providing an array to the `$fail` callback injected into Closure based validation rules. However, you should now provide the key as the first argument and the failure message as the second argument:

```
Validator::make([
    'foo' => 'string',
    'bar' => [function ($attribute, $value, $fail) {
        $fail('foo', 'Something went wrong!');
    }],
]);
```

Form Request After Method

Likelihood Of Impact: Very Low

Within form requests, the `after` method is now reserved by Laravel. If your form requests define an `after` method, the method should be renamed or modified to utilize the new "after validation" feature of Laravel's form requests.

Miscellaneous

We also encourage you to view the changes in the [laravel/laravel GitHub repository](#). While many of these changes are not required, you may wish to keep these files in sync with your application. Some of these changes will be covered in this upgrade guide, but others, such as changes to configuration files or comments, will not be.

You can easily view the changes with the [GitHub comparison tool](#) and choose which updates are important to you. However, many of the changes shown by the GitHub comparison tool are due to our organization's adoption of PHP native types. These changes are backwards compatible and the adoption of them during the migration to Laravel 10 is optional.

Contribution Guide

- [Bug Reports](#)
- [Support Questions](#)
- [Core Development Discussion](#)
- [Which Branch?](#)
- [Compiled Assets](#)
- [Security Vulnerabilities](#)
- [Coding Style](#)
 - [PHPDoc](#)
 - [StyleCI](#)
- [Code of Conduct](#)

Bug Reports

To encourage active collaboration, Laravel strongly encourages pull requests, not just bug reports. Pull requests will only be reviewed when marked as "ready for review" (not in the "draft" state) and all tests for new features are passing. Lingering, non-active pull requests left in the "draft" state will be closed after a few days.

However, if you file a bug report, your issue should contain a title and a clear description of the issue. You should also include as much relevant information as possible and a code sample that demonstrates the issue. The goal of a bug report is to make it easy for yourself - and others - to replicate the bug and develop a fix.

Remember, bug reports are created in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the bug report will automatically see any activity or that others will jump to fix it. Creating a bug report serves to help yourself and others start on the path of fixing the problem. If you want to chip in, you can help out by fixing [any bugs listed in our issue trackers](#). You must be authenticated with GitHub to view all of Laravel's issues.

If you notice improper DocBlock, PHPStan, or IDE warnings while using Laravel, do not create a GitHub issue. Instead, please submit a pull request to fix the problem.

The Laravel source code is managed on GitHub, and there are repositories for each of the Laravel projects:

- [Laravel Application](#)
- [Laravel Art](#)
- [Laravel Documentation](#)
- [Laravel Dusk](#)
- [Laravel Cashier Stripe](#)
- [Laravel Cashier Paddle](#)
- [Laravel Echo](#)
- [Laravel Envoy](#)
- [Laravel Folio](#)
- [Laravel Framework](#)
- [Laravel Homestead](#)
- [Laravel Homestead Build Scripts](#)
- [Laravel Horizon](#)
- [Laravel Jetstream](#)
- [Laravel Passport](#)
- [Laravel Pennant](#)
- [Laravel Pint](#)
- [Laravel Prompts](#)
- [Laravel Sail](#)
- [Laravel Sanctum](#)
- [Laravel Scout](#)
- [Laravel Socialite](#)

- [Laravel Telescope](#)
- [Laravel Website](#)

Support Questions

Laravel's GitHub issue trackers are not intended to provide Laravel help or support. Instead, use one of the following channels:

- [GitHub Discussions](#)
- [Laracasts Forums](#)
- [Laravel.io Forums](#)
- [StackOverflow](#)
- [Discord](#)
- [Larachat](#)
- [IRC](#)

Core Development Discussion

You may propose new features or improvements of existing Laravel behavior in the Laravel framework repository's [GitHub discussion board](#). If you propose a new feature, please be willing to implement at least some of the code that would be needed to complete the feature.

Informal discussion regarding bugs, new features, and implementation of existing features takes place in the [#internals](#) channel of the [Laravel Discord server](#). Taylor Otwell, the maintainer of Laravel, is typically present in the channel on weekdays from 8am-5pm (UTC-06:00 or America/Chicago), and sporadically present in the channel at other times.

Which Branch?

All bug fixes should be sent to the latest version that supports bug fixes (currently [10.x](#)). Bug fixes should **never** be sent to the [master](#) branch unless they fix features that exist only in the upcoming release.

Minor features that are **fully backward compatible** with the current release may be sent to the latest stable branch (currently [10.x](#)).

Major new features or features with breaking changes should always be sent to the [master](#) branch, which contains the upcoming release.

Compiled Assets

If you are submitting a change that will affect a compiled file, such as most of the files in [resources/css](#) or [resources/js](#) of the [laravel/laravel](#) repository, do not commit the compiled files. Due to their large size, they cannot realistically be reviewed by a maintainer. This could be exploited as a way to inject malicious code into Laravel. In order to defensively prevent this, all compiled files will be generated and committed by Laravel maintainers.

Security Vulnerabilities

If you discover a security vulnerability within Laravel, please send an email to Taylor Otwell at taylor@laravel.com. All security vulnerabilities will be promptly addressed.

Coding Style

Laravel follows the [PSR-2](#) coding standard and the [PSR-4](#) autoloading standard.

PHPDoc

Below is an example of a valid Laravel documentation block. Note that the [@param](#) attribute is followed by two spaces, the argument type, two more spaces, and finally the variable name:

```
/**
 * Register a binding with the container.
 *
 * @param string|array $abstract
 * @param \Closure|string|null $concrete
 * @param bool $shared
 * @return void
 *
 * @throws \Exception
 */
public function bind($abstract, $concrete = null, $shared = false)
{
    // ...
}
```

When the `@param` or `@return` attributes are redundant due to the use of native types, they can be removed:

```
/**
 * Execute the job.
 */
public function handle(AudioProcessor $processor): void
{
    //
}
```

However, when the native type is generic, please specify the generic type through the use of the `@param` or `@return` attributes:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file'),
    ];
}
```

StyleCI

Don't worry if your code styling isn't perfect! [StyleCI](#) will automatically merge any style fixes into the Laravel repository after pull requests are merged. This allows us to focus on the content of the contribution and not the code style.

Code of Conduct

The Laravel code of conduct is derived from the Ruby code of conduct. Any violations of the code of conduct may be reported to Taylor Otwell (taylor@laravel.com):

- Participants will be tolerant of opposing views.
- Participants must ensure that their language and actions are free of personal attacks and disparaging personal remarks.
- When interpreting the words and actions of others, participants should always assume good intentions.

- Behavior that can be reasonably considered harassment will not be tolerated.

Chapter 2

Getting Started

Installation

- [Meet Laravel](#)
 - [Why Laravel?](#)
- [Creating a Laravel Project](#)
- [Initial Configuration](#)
 - [Environment Based Configuration](#)
 - [Databases and Migrations](#)
 - [Directory Configuration](#)
- [Docker Installation Using Sail](#)
 - [Sail on macOS](#)
 - [Sail on Windows](#)
 - [Sail on Linux](#)
 - [Choosing Your Sail Services](#)
- [IDE Support](#)
- [Laravel and AI](#)
 - [Installing Laravel Boost](#)
- [Next Steps](#)
 - [Laravel the Full Stack Framework](#)
 - [Laravel the API Backend](#)

Meet Laravel

Laravel is a web application framework with expressive, elegant syntax. A web framework provides a structure and starting point for creating your application, allowing you to focus on creating something amazing while we sweat the details.

Laravel strives to provide an amazing developer experience while providing powerful features such as thorough dependency injection, an expressive database abstraction layer, queues and scheduled jobs, unit and integration testing, and more.

Whether you are new to PHP web frameworks or have years of experience, Laravel is a framework that can grow with you. We'll help you take your first steps as a web developer or give you a boost as you take your expertise to the next level. We can't wait to see what you build.



New to Laravel? Check out the [Laravel Bootcamp](#) for a hands-on tour of the framework while we walk you through building your first Laravel application.

Why Laravel?

There are a variety of tools and frameworks available to you when building a web application. However, we believe Laravel is the best choice for building modern, full-stack web applications.

A Progressive Framework

We like to call Laravel a "progressive" framework. By that, we mean that Laravel grows with you. If you're just taking your first steps into web development, Laravel's vast library of documentation, guides, and [video tutorials](#) will help you learn the ropes without becoming overwhelmed.

If you're a senior developer, Laravel gives you robust tools for [dependency injection](#), [unit testing](#), [queues](#), [real-time events](#), and more. Laravel is fine-tuned for building professional web applications and ready to handle enterprise work loads.

A Scalable Framework

Laravel is incredibly scalable. Thanks to the scaling-friendly nature of PHP and Laravel's built-in support for fast, distributed cache systems like Redis, horizontal scaling with Laravel is a breeze. In fact, Laravel applications have been easily scaled to handle hundreds of millions of requests per month.

Need extreme scaling? Platforms like [Laravel Vapor](#) allow you to run your Laravel application at nearly limitless scale on AWS's latest serverless technology.

A Community Framework

Laravel combines the best packages in the PHP ecosystem to offer the most robust and developer friendly framework available. In addition, thousands of talented developers from around the world have [contributed to the framework](#). Who knows, maybe you'll even become a Laravel contributor.

Creating a Laravel Project

Before creating your first Laravel project, make sure that your local machine has PHP and [Composer](#) installed. If you are developing on macOS, PHP and Composer can be installed in minutes via [Laravel Herd](#). In addition, we recommend [installing Node and NPM](#).

After you have installed PHP and Composer, you may create a new Laravel project via Composer's `create-project` command:

```
composer create-project "laravel/laravel:^10.0" example-app
```

Or, you may create new Laravel projects by globally installing [the Laravel installer](#) via Composer:

```
composer global require laravel/installer
laravel new example-app
```

Once the project has been created, start Laravel's local development server using Laravel Artisan's `serve` command:

```
cd example-app
php artisan serve
```

Once you have started the Artisan development server, your application will be accessible in your web browser at <http://localhost:8000>. Next, you're ready to [start taking your next steps into the Laravel ecosystem](#). Of course, you may also want to [configure a database](#).



If you would like a head start when developing your Laravel application, consider using one of our [starter kits](#). Laravel's starter kits provide backend and frontend authentication scaffolding for your new Laravel application.

Initial Configuration

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.

Laravel needs almost no additional configuration out of the box. You are free to get started developing! However, you may wish to review the `config/app.php` file and its documentation. It contains several options such as `timezone` and

`locale` that you may wish to change according to your application.

Environment Based Configuration

Since many of Laravel's configuration option values may vary depending on whether your application is running on your local machine or on a production web server, many important configuration values are defined using the `.env` file that exists at the root of your application.

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.



For more information about the `.env` file and environment based configuration, check out the full [configuration documentation](#).

Databases and Migrations

Now that you have created your Laravel application, you probably want to store some data in a database. By default, your application's `.env` configuration file specifies that Laravel will be interacting with a MySQL database and will access the database at `127.0.0.1`.



If you are developing on macOS and need to install MySQL, Postgres, or Redis locally, consider using [DBngin](#).

If you do not want to install MySQL or Postgres on your local machine, you can always use a [SQLite](#) database. SQLite is a small, fast, self-contained database engine. To get started, update your `.env` configuration file to use Laravel's `sqlite` database driver. You may remove the other database configuration options:

```
+ DB_CONNECTION=sqlite
- DB_CONNECTION=mysql
- DB_HOST=127.0.0.1
- DB_PORT=3306
- DB_DATABASE=laravel
- DB_USERNAME=root
- DB_PASSWORD=
```

Once you have configured your SQLite database, you may run your application's [database migrations](#), which will create your application's database tables:

```
php artisan migrate
```

If an SQLite database does not exist for your application, Laravel will ask you if you would like the database to be created. Typically, the SQLite database file will be created at `database/database.sqlite`.

Directory Configuration

Laravel should always be served out of the root of the "web directory" configured for your web server. You should not attempt to serve a Laravel application out of a subdirectory of the "web directory". Attempting to do so could expose sensitive files present within your application.

Docker Installation Using Sail

We want it to be as easy as possible to get started with Laravel regardless of your preferred operating system. So, there are a variety of options for developing and running a Laravel project on your local machine. While you may wish to explore these options at a later time, Laravel provides [Sail](#), a built-in solution for running your Laravel project using [Docker](#).

Docker is a tool for running applications and services in small, light-weight "containers" which do not interfere with your local machine's installed software or configuration. This means you don't have to worry about configuring or setting up complicated development tools such as web servers and databases on your local machine. To get started, you only need to install [Docker Desktop](#).

Laravel Sail is a light-weight command-line interface for interacting with Laravel's default Docker configuration. Sail provides a great starting point for building a Laravel application using PHP, MySQL, and Redis without requiring prior Docker experience.



Already a Docker expert? Don't worry! Everything about Sail can be customized using the `docker-compose.yml` file included with Laravel.

Sail on macOS

If you're developing on a Mac and [Docker Desktop](#) is already installed, you can use a simple terminal command to create a new Laravel project. For example, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

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

Of course, you can change "example-app" in this URL to anything you like - just make sure the application name only contains alpha-numeric characters, dashes, and underscores. The Laravel application's directory will be created within the directory you execute the command from.

Sail installation may take several minutes while Sail's application containers are built on your local machine.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app
./vendor/bin/sail up
```

Once the application's Docker containers have been started, you can access the application in your web browser at: <http://localhost>.



To continue learning more about Laravel Sail, review its [complete documentation](#).

Sail on Windows

Before we create a new Laravel application on your Windows machine, make sure to install [Docker Desktop](#). Next, you should ensure that Windows Subsystem for Linux 2 (WSL2) is installed and enabled. WSL allows you to run Linux binary executables natively on Windows 10. Information on how to install and enable WSL2 can be found within Microsoft's [developer environment documentation](#).



After installing and enabling WSL2, you should ensure that Docker Desktop is [configured to use the WSL2 backend](#).

Next, you are ready to create your first Laravel project. Launch [Windows Terminal](#) and begin a new terminal session for your WSL2 Linux operating system. Next, you can use a simple terminal command to create a new Laravel project. For example, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

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

Of course, you can change "example-app" in this URL to anything you like - just make sure the application name only contains alpha-numeric characters, dashes, and underscores. The Laravel application's directory will be created within the directory you execute the command from.

Sail installation may take several minutes while Sail's application containers are built on your local machine.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app
./vendor/bin/sail up
```

Once the application's Docker containers have been started, you can access the application in your web browser at: <http://localhost>.



To continue learning more about Laravel Sail, review its [complete documentation](#).

Developing Within WSL2

Of course, you will need to be able to modify the Laravel application files that were created within your WSL2 installation. To accomplish this, we recommend using Microsoft's [Visual Studio Code](#) editor and their first-party extension for [Remote Development](#).

Once these tools are installed, you may open any Laravel project by executing the `code .` command from your application's root directory using Windows Terminal.

Sail on Linux

If you're developing on Linux and [Docker Compose](#) is already installed, you can use a simple terminal command to create a new Laravel project.

First, if you are using Docker Desktop for Linux, you should execute the following command. If you are not using Docker Desktop for Linux, you may skip this step:

```
docker context use default
```

Then, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

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

Of course, you can change "example-app" in this URL to anything you like - just make sure the application name only contains alpha-numeric characters, dashes, and underscores. The Laravel application's directory will be created within the directory you execute the command from.

Sail installation may take several minutes while Sail's application containers are built on your local machine.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app
./vendor/bin/sail up
```

Once the application's Docker containers have been started, you can access the application in your web browser at: <http://localhost>.



To continue learning more about Laravel Sail, review its [complete documentation](#).

Choosing Your Sail Services

When creating a new Laravel application via Sail, you may use the `with` query string variable to choose which services should be configured in your new application's `docker-compose.yml` file. Available services include `mysql`, `pgsql`, `mariadb`, `redis`, `memcached`, `elasticsearch`, `typesense`, `minio`, `selenium`, and `mailpit`:

```
curl -s "https://laravel.build/example-app?with=mysql,redis" | bash
```

If you do not specify which services you would like configured, a default stack of `mysql`, `redis`, `elasticsearch`, `mailpit`, and `selenium` will be configured.

You may instruct Sail to install a default `Devcontainer` by adding the `devcontainer` parameter to the URL:

```
curl -s "https://laravel.build/example-app?with=mysql,redis&devcontainer" | bash
```

IDE Support

You are free to use any code editor you wish when developing Laravel applications; however, [PhpStorm](#) offers extensive support for Laravel and its ecosystem, including [Laravel Pint](#).

In addition, the community maintained [Laravel Idea](#) PhpStorm plugin offers a variety of helpful IDE augmentations, including code generation, Eloquent syntax completion, validation rule completion, and more.

Laravel and AI

[Laravel Boost](#) is a powerful tool that bridges the gap between AI coding agents and Laravel applications. Boost provides AI agents with Laravel-specific context, tools, and guidelines so they can generate more accurate, version-specific code that follows Laravel conventions.

When you install Boost in your Laravel application, AI agents gain access to over 15 specialized tools including the ability to know which packages you are using, query your database, search the Laravel documentation, read browser logs, generate

tests, and execute code via Tinker.

In addition, Boost gives AI agents access to over 17,000 pieces of vectorized Laravel ecosystem documentation, specific to your installed package versions. This means agents can provide guidance targeted to the exact versions your project uses.

Boost also includes Laravel-maintained AI guidelines that nudge agents to follow framework conventions, write appropriate tests, and avoid common pitfalls when generating Laravel code.

Installing Laravel Boost

Boost can be installed in Laravel 10, 11, and 12 applications running PHP 8.1 or higher. To get started, install Boost as a development dependency:

```
composer require laravel/boost --dev
```

Once installed, run the interactive installer:

```
php artisan boost:install
```

The installer will auto-detect your IDE and AI agents, allowing you to opt into the features that make sense for your project. Boost respects existing project conventions and doesn't force opinionated style rules by default.



To learn more about Boost, check out the [Laravel Boost repository on GitHub](#).

Next Steps

Now that you have created your Laravel project, you may be wondering what to learn next. First, we strongly recommend becoming familiar with how Laravel works by reading the following documentation:

- [Request Lifecycle](#)
- [Configuration](#)
- [Directory Structure](#)
- [Frontend](#)
- [Service Container](#)
- [Facades](#)

How you want to use Laravel will also dictate the next steps on your journey. There are a variety of ways to use Laravel, and we'll explore two primary use cases for the framework below.



New to Laravel? Check out the [Laravel Bootcamp](#) for a hands-on tour of the framework while we walk you through building your first Laravel application.

Laravel the Full Stack Framework

Laravel may serve as a full stack framework. By "full stack" framework we mean that you are going to use Laravel to route requests to your application and render your frontend via [Blade templates](#) or a single-page application hybrid technology like [Inertia](#). This is the most common way to use the Laravel framework, and, in our opinion, the most productive way to use Laravel.

If this is how you plan to use Laravel, you may want to check out our documentation on [frontend development](#), [routing](#), [views](#), or the [Eloquent ORM](#). In addition, you might be interested in learning about community packages like [Livewire](#) and

Inertia. These packages allow you to use Laravel as a full-stack framework while enjoying many of the UI benefits provided by single-page JavaScript applications.

If you are using Laravel as a full stack framework, we also strongly encourage you to learn how to compile your application's CSS and JavaScript using [Vite](#).



If you want to get a head start building your application, check out one of our official [application starter kits](#).

Laravel the API Backend

Laravel may also serve as an API backend to a JavaScript single-page application or mobile application. For example, you might use Laravel as an API backend for your [Next.js](#) application. In this context, you may use Laravel to provide [authentication](#) and data storage / retrieval for your application, while also taking advantage of Laravel's powerful services such as queues, emails, notifications, and more.

If this is how you plan to use Laravel, you may want to check out our documentation on [routing](#), [Laravel Sanctum](#), and the [Eloquent ORM](#).



Need a head start scaffolding your Laravel backend and Next.js frontend? Laravel Breeze offers an [API stack](#) as well as a [Next.js frontend implementation](#) so you can get started in minutes.

Configuration

- [Introduction](#)
- [Environment Configuration](#)
 - [Environment Variable Types](#)
 - [Retrieving Environment Configuration](#)
 - [Determining the Current Environment](#)
 - [Encrypting Environment Files](#)
- [Accessing Configuration Values](#)
- [Configuration Caching](#)
- [Debug Mode](#)
- [Maintenance Mode](#)

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.

Application Overview

In a hurry? You can get a quick overview of your application's configuration, drivers, and environment via the `about` Artisan command:

```
php artisan about
```

If you're only interested in a particular section of the application overview output, you may filter for that section using the `--only` option:

```
php artisan about --only=environment
```

Or, to explore a specific configuration file's values in detail, you may use the `config:show` Artisan command:

```
php artisan config:show database
```

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](#) 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

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.

However, it is possible to encrypt your environment file using Laravel's built-in [environment encryption](#). Encrypted environment files may be placed in source control safely.

Additional Environment Files

Before loading your application's environment variables, Laravel determines if an `APP_ENV` environment variable has been externally provided or if the `--env` CLI argument has been specified. If so, Laravel will attempt to load an `.env.[APP_ENV]` file if it exists. If it does not exist, the default `.env` file will be loaded.

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:

<code>.env</code> Value	<code>env()</code> Value
<code>true</code>	(bool) true
<code>(true)</code>	(bool) true
<code>false</code>	(bool) false
<code>(false)</code>	(bool) false
<code>empty</code>	(string) ''
<code>(empty)</code>	(string) ''
<code>null</code>	(null) null
<code>(null)</code>	(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

All of the variables listed in the `.env` file will be loaded into the `$_ENV` PHP super-global when your application receives a request. However, you may use the `env` function 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 function:

```
'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

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`:

```
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.

Encrypting Environment Files

Unencrypted environment files should never be stored in source control. However, Laravel allows you to encrypt your environment files so that they may safely be added to source control with the rest of your application.

Encryption

To encrypt an environment file, you may use the `env:encrypt` command:

```
php artisan env:encrypt
```

Running the `env:encrypt` command will encrypt your `.env` file and place the encrypted contents in an `.env.encrypted` file. The decryption key is presented in the output of the command and should be stored in a secure password manager. If you would like to provide your own encryption key you may use the `--key` option when invoking the command:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```



The length of the key provided should match the key length required by the encryption cipher being used. By default, Laravel will use the `AES-256-CBC` cipher which requires a 32 character key. You are free to use any cipher supported by Laravel's `encrypter` by passing the `--cipher` option when invoking the command.

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be encrypted by providing the environment name via the `--env` option:

```
php artisan env:encrypt --env=staging
```

Decryption

To decrypt an environment file, you may use the `env:decrypt` command. This command requires a decryption key, which Laravel will retrieve from the `LARAVEL_ENV_ENCRYPTION_KEY` environment variable:

```
php artisan env:decrypt
```

Or, the key may be provided directly to the command via the `--key` option:

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvobsZN1aF
```

When the `env:decrypt` command is invoked, Laravel will decrypt the contents of the `.env.encrypted` file and place the decrypted contents in the `.env` file.

The `--cipher` option may be provided to the `env:decrypt` command in order to use a custom encryption cipher:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be decrypted by providing the environment name via the `--env` option:

```
php artisan env:decrypt --env=staging
```

In order to overwrite an existing environment file, you may provide the `--force` option to the `env:decrypt` command:

```
php artisan env:decrypt --force
```

Accessing Configuration Values

You may easily access your configuration values using the `Config` facade or global `config` 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:

```
use Illuminate\Support\Facades\Config;

$value = Config::get('app.timezone');

$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, you may invoke the `Config` facade's `set` method or pass an array to the `config` function:

```
Config::set('app.timezone', 'America/Chicago');

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

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.

Once the configuration has been cached, your application's `.env` file will not be loaded by the framework during requests or Artisan commands; therefore, the `env` function will only return external, system level environment variables.

For this reason, you should ensure you are only calling the `env` function from within your application's configuration (`config`) files. You can see many examples of this by examining Laravel's default configuration files. Configuration values may be accessed from anywhere in your application using the `config` function [described above](#).

The `config:clear` command may be used to purge the cached configuration:

```
php artisan config:clear
```



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

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

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\RuntimeException` 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

To allow maintenance mode to be bypassed using a secret token, 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
```

If you would like Laravel to generate the secret token for you, you may use the `with-secret` option. The secret will be displayed to you once the application is in maintenance mode:

```
php artisan down --with-secret
```

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.



Your maintenance mode secret should typically consist of alpha-numeric characters and, optionally, dashes. You should avoid using characters that have special meaning in URLs such as `?` or `&`.

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

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

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 and Queues

While your application is in maintenance mode, no queued jobs will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

Alternatives to Maintenance Mode

Since maintenance mode requires your application to have several seconds of downtime, consider alternatives like [Laravel Vapor](#) and [Envoyer](#) to accomplish zero-downtime deployment with Laravel.

Directory Structure

- [Introduction](#)
- [The Root Directory](#)
 - [The app Directory](#)
 - [The bootstrap Directory](#)
 - [The config Directory](#)
 - [The database Directory](#)
 - [The public Directory](#)
 - [The resources Directory](#)
 - [The routes Directory](#)
 - [The storage Directory](#)
 - [The tests Directory](#)
 - [The vendor Directory](#)
- [The App Directory](#)
 - [The Broadcasting Directory](#)
 - [The Console Directory](#)
 - [The Events Directory](#)
 - [The Exceptions Directory](#)
 - [The Http Directory](#)
 - [The Jobs Directory](#)
 - [The Listeners Directory](#)
 - [The Mail Directory](#)
 - [The Models Directory](#)
 - [The Notifications Directory](#)
 - [The Policies Directory](#)
 - [The Providers Directory](#)
 - [The Rules Directory](#)

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.



New to Laravel? Check out the [Laravel Bootcamp](#) for a hands-on tour of the framework while we walk you through building your first Laravel application.

The Root Directory

The 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

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

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

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

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

The `resources` directory contains your [views](#) as well as your raw, un-compiled assets such as CSS or JavaScript.

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 all 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](#) 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](#) channels that your application supports.

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 location of the `storage` directory can be modified via the `LARAVEL_STORAGE_PATH` environment variable.

The Tests Directory

The `tests` directory contains your automated tests. Example [PHPUnit](#) 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

The `vendor` directory contains your [Composer](#) dependencies.

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](#).

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

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](#).

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](#) are defined.

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](#). 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

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

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

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](#) 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](#).

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. 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

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 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

The `Models` directory contains all of your Eloquent model classes. 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

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 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

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 for your application. Policies are used to determine if a user can perform a given action against a resource.

The Providers Directory

The `Providers` directory contains all of the service 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

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.

Frontend

- [Introduction](#)
- [Using PHP](#)
 - [PHP and Blade](#)
 - [Livewire](#)
 - [Starter Kits](#)
- [Using Vue / React](#)
 - [Inertia](#)
 - [Starter Kits](#)
- [Bundling Assets](#)

Introduction

Laravel is a backend framework that provides all of the features you need to build modern web applications, such as [routing](#), [validation](#), [caching](#), [queues](#), [file storage](#), and more. However, we believe it's important to offer developers a beautiful full-stack experience, including powerful approaches for building your application's frontend.

There are two primary ways to tackle frontend development when building an application with Laravel, and which approach you choose is determined by whether you would like to build your frontend by leveraging PHP or by using JavaScript frameworks such as Vue and React. We'll discuss both of these options below so that you can make an informed decision regarding the best approach to frontend development for your application.

Using PHP

PHP and Blade

In the past, most PHP applications rendered HTML to the browser using simple HTML templates interspersed with PHP `echo` statements which render data that was retrieved from a database during the request:

```
<div>
    <?php foreach ($users as $user): ?>
        Hello, <?php echo $user->name; ?> <br />
    <?php endforeach; ?>
</div>
```

In Laravel, this approach to rendering HTML can still be achieved using [views](#) and [Blade](#). Blade is an extremely light-weight templating language that provides convenient, short syntax for displaying data, iterating over data, and more:

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

When building applications in this fashion, form submissions and other page interactions typically receive an entirely new HTML document from the server and the entire page is re-rendered by the browser. Even today, many applications may be perfectly suited to having their frontends constructed in this way using simple Blade templates.

Growing Expectations

However, as user expectations regarding web applications have matured, many developers have found the need to build more dynamic frontends with interactions that feel more polished. In light of this, some developers choose to begin

building their application's frontend using JavaScript frameworks such as Vue and React.

Others, preferring to stick with the backend language they are comfortable with, have developed solutions that allow the construction of modern web application UIs while still primarily utilizing their backend language of choice. For example, in the [Rails](#) ecosystem, this has spurred the creation of libraries such as [Turbo Hotwire](#), and [Stimulus](#).

Within the Laravel ecosystem, the need to create modern, dynamic frontends by primarily using PHP has led to the creation of [Laravel Livewire](#) and [Alpine.js](#).

Livewire

[Laravel Livewire](#) is a framework for building Laravel powered frontends that feel dynamic, modern, and alive just like frontends built with modern JavaScript frameworks like Vue and React.

When using Livewire, you will create Livewire "components" that render a discrete portion of your UI and expose methods and data that can be invoked and interacted with from your application's frontend. For example, a simple "Counter" component might look like the following:

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
    {
        return view('livewire.counter');
    }
}
```

And, the corresponding template for the counter would be written like so:

```
<div>
    <button wire:click="increment">+</button>
    <h1>{{ $count }}</h1>
</div>
```

As you can see, Livewire enables you to write new HTML attributes such as `wire:click` that connect your Laravel application's frontend and backend. In addition, you can render your component's current state using simple Blade expressions.

For many, Livewire has revolutionized frontend development with Laravel, allowing them to stay within the comfort of Laravel while constructing modern, dynamic web applications. Typically, developers using Livewire will also utilize [Alpine.js](#) to "sprinkle" JavaScript onto their frontend only where it is needed, such as in order to render a dialog window.

If you're new to Laravel, we recommend getting familiar with the basic usage of [views](#) and [Blade](#). Then, consult the official [Laravel Livewire documentation](#) to learn how to take your application to the next level with interactive Livewire components.

Starter Kits

If you would like to build your frontend using PHP and Livewire, you can leverage our Breeze or Jetstream [starter kits](#) to jump-start your application's development. Both of these starter kits scaffold your application's backend and frontend authentication flow using [Blade](#) and [Tailwind](#) so that you can simply start building your next big idea.

Using Vue / React

Although it's possible to build modern frontends using Laravel and Livewire, many developers still prefer to leverage the power of a JavaScript framework like Vue or React. This allows developers to take advantage of the rich ecosystem of JavaScript packages and tools available via NPM.

However, without additional tooling, pairing Laravel with Vue or React would leave us needing to solve a variety of complicated problems such as client-side routing, data hydration, and authentication. Client-side routing is often simplified by using opinionated Vue / React frameworks such as [Nuxt](#) and [Next](#); however, data hydration and authentication remain complicated and cumbersome problems to solve when pairing a backend framework like Laravel with these frontend frameworks.

In addition, developers are left maintaining two separate code repositories, often needing to coordinate maintenance, releases, and deployments across both repositories. While these problems are not insurmountable, we don't believe it's a productive or enjoyable way to develop applications.

Inertia

Thankfully, Laravel offers the best of both worlds. [Inertia](#) bridges the gap between your Laravel application and your modern Vue or React frontend, allowing you to build full-fledged, modern frontends using Vue or React while leveraging Laravel routes and controllers for routing, data hydration, and authentication — all within a single code repository. With this approach, you can enjoy the full power of both Laravel and Vue / React without crippling the capabilities of either tool.

After installing Inertia into your Laravel application, you will write routes and controllers like normal. However, instead of returning a Blade template from your controller, you will return an Inertia page:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Inertia\Inertia;
use Inertia\Response;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): Response
    {
        return Inertia::render('Users/Profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

An Inertia page corresponds to a Vue or React component, typically stored within the `resources/js/Pages` directory of your application. The data given to the page via the `Inertia::render` method will be used to hydrate the "props" of the page component:

```

<script setup>
import Layout from '@/Layouts/Authenticated.vue';
import { Head } from '@inertiajs/vue3';

const props = defineProps(['user']);
</script>

<template>
    <Head title="User Profile" />

    <Layout>
        <template #header>
            <h2 class="font-semibold text-xl text-gray-800 leading-tight">
                Profile
            </h2>
        </template>

        <div class="py-12">
            Hello, {{ user.name }}
        </div>
    </Layout>
</template>

```

As you can see, Inertia allows you to leverage the full power of Vue or React when building your frontend, while providing a light-weight bridge between your Laravel powered backend and your JavaScript powered frontend.

Server-Side Rendering

If you're concerned about diving into Inertia because your application requires server-side rendering, don't worry. Inertia offers [server-side rendering support](#). And, when deploying your application via [Laravel Forge](#), it's a breeze to ensure that Inertia's server-side rendering process is always running.

Starter Kits

If you would like to build your frontend using Inertia and Vue / React, you can leverage our Breeze or Jetstream [starter kits](#) to jump-start your application's development. Both of these starter kits scaffold your application's backend and frontend authentication flow using Inertia, Vue / React, [Tailwind](#), and [Vite](#) so that you can start building your next big idea.

Bundling Assets

Regardless of whether you choose to develop your frontend using Blade and Livewire or Vue / React and Inertia, you will likely need to bundle your application's CSS into production ready assets. Of course, if you choose to build your application's frontend with Vue or React, you will also need to bundle your components into browser ready JavaScript assets.

By default, Laravel utilizes [Vite](#) to bundle your assets. Vite provides lightning-fast build times and near instantaneous Hot Module Replacement (HMR) during local development. In all new Laravel applications, including those using our [starter kits](#), you will find a `vite.config.js` file that loads our light-weight Laravel Vite plugin that makes Vite a joy to use with Laravel applications.

The fastest way to get started with Laravel and Vite is by beginning your application's development using [Laravel Breeze](#), our simplest starter kit that jump-starts your application by providing frontend and backend authentication scaffolding.



For more detailed documentation on utilizing Vite with Laravel, please see our [dedicated documentation on bundling and compiling your assets](#).

Starter Kits

- [Introduction](#)
- [Laravel Breeze](#)
 - [Installation](#)
 - [Breeze and Blade](#)
 - [Breeze and Livewire](#)
 - [Breeze and React / Vue](#)
 - [Breeze and Next.js / API](#)
- [Laravel Jetstream](#)

Introduction

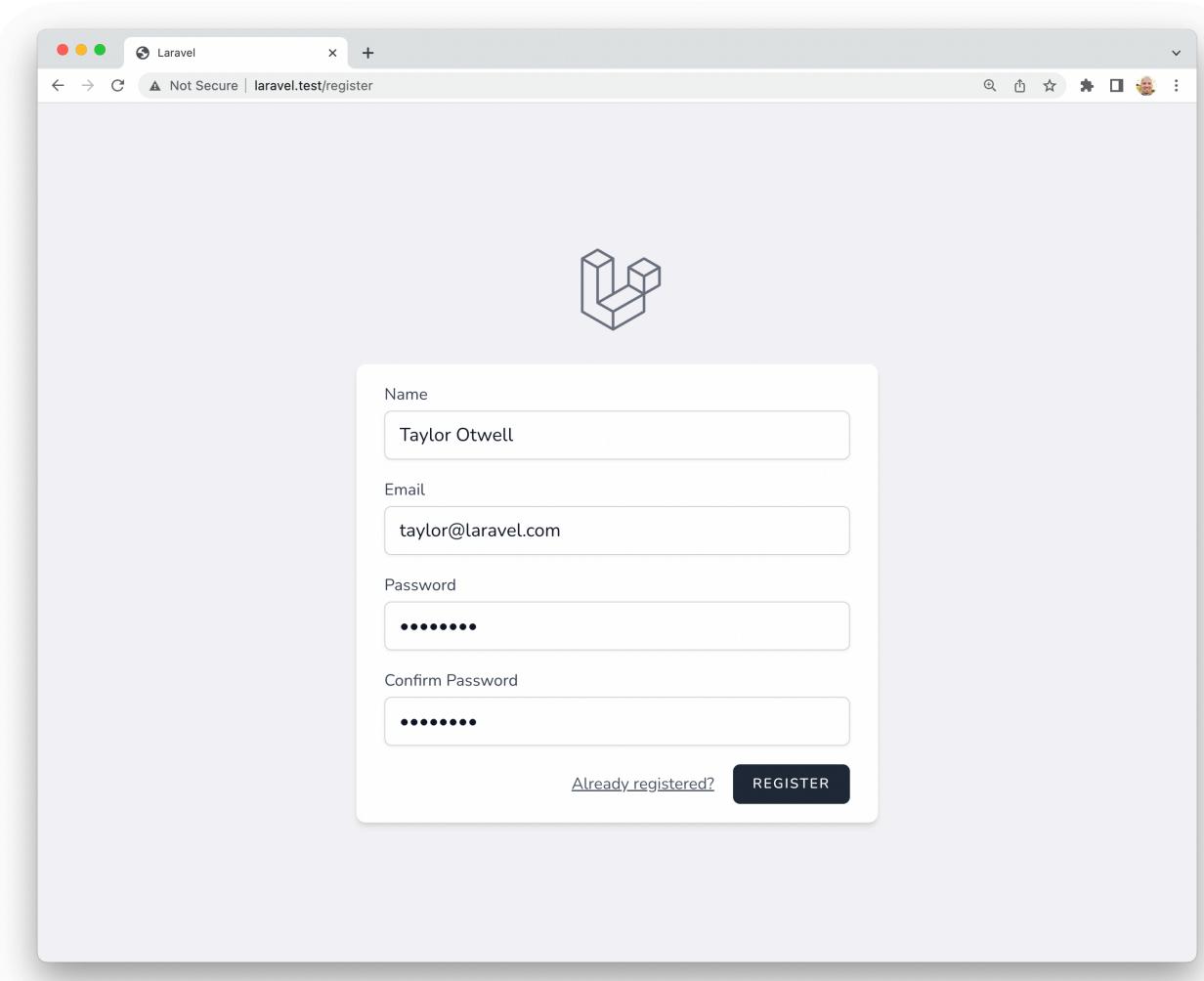
To give you a head start building your new Laravel application, we are happy to offer authentication and application starter kits. These kits automatically scaffold your application with the routes, controllers, and views you need to register and authenticate your application's users.

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

[Laravel Breeze](#) is a minimal, simple implementation of all of Laravel's [authentication features](#), including login, registration, password reset, email verification, and password confirmation. In addition, Breeze includes a simple "profile" page where the user may update their name, email address, and password.

Laravel Breeze's default view layer is made up of simple [Blade templates](#) styled with [Tailwind CSS](#). Additionally, Breeze provides scaffolding options based on [Livewire](#) or [Inertia](#), with the choice of using Vue or React for the Inertia-based scaffolding.



Laravel Bootcamp

If you're new to Laravel, feel free to jump into the [Laravel Bootcamp](#). The Laravel Bootcamp will walk you through building your first Laravel application using Breeze. It's a great way to get a tour of everything that Laravel and Breeze have to offer.

Installation

First, you should [create a new Laravel application](#), configure your database, and run your [database migrations](#). 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.

The `breeze:install` command will prompt you for your preferred frontend stack and testing framework:

```
php artisan breeze:install  
php artisan migrate  
npm install  
npm run dev
```

Breeze and Blade

The default Breeze "stack" is the Blade stack, which utilizes simple [Blade templates](#) to render your application's frontend. The Blade stack may be installed by invoking the `breeze:install` command with no other additional arguments and selecting the Blade frontend stack. After Breeze's scaffolding is installed, you should also compile your application's frontend assets:

```
php artisan breeze:install

php artisan migrate
npm install
npm run dev
```

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 Laravel's [Vite documentation](#).

Breeze and Livewire

Laravel Breeze also offers [Livewire](#) scaffolding. Livewire is a powerful way of building dynamic, reactive, front-end UIs using just PHP.

Livewire is a great fit for teams that primarily use Blade templates and are looking for a simpler alternative to JavaScript-driven SPA frameworks like Vue and React.

To use the Livewire stack, you may select the Livewire frontend stack when executing the `breeze:install` Artisan command. After Breeze's scaffolding is installed, you should run your database migrations:

```
php artisan breeze:install

php artisan migrate
```

Breeze and React / Vue

Laravel Breeze also offers React and Vue scaffolding via an [Inertia](#) frontend implementation. Inertia allows you to build modern, single-page React and Vue applications using classic server-side routing and controllers.

Inertia lets you enjoy the frontend power of React and Vue combined with the incredible backend productivity of Laravel and lightning-fast [Vite](#) compilation. To use an Inertia stack, you may select the Vue or React frontend stacks when executing the `breeze:install` Artisan command.

When selecting the Vue or React frontend stack, the Breeze installer will also prompt you to determine if you would like [Inertia SSR](#) or TypeScript support. After Breeze's scaffolding is installed, you should also compile your application's frontend assets:

```
php artisan breeze:install

php artisan migrate
npm install
npm run dev
```

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.

Breeze and Next.js / API

Laravel Breeze can also scaffold an authentication API that is ready to authenticate modern JavaScript applications such as those powered by [Next](#), [Nuxt](#), and others. To get started, select the API stack as your desired stack when executing the `breeze:install` Artisan command:

```
php artisan breeze:install

php artisan migrate
```

During installation, Breeze will add a `FRONTEND_URL` environment variable to your application's `.env` file. This URL should be the URL of your JavaScript application. This will typically be `http://localhost:3000` during local development. In addition, you should ensure that your `APP_URL` is set to `http://localhost:8000`, which is the default URL used by the `serve` Artisan command.

Next.js Reference Implementation

Finally, you are ready to pair this backend with the frontend of your choice. A Next reference implementation of the Breeze frontend is [available on GitHub](#). This frontend is maintained by Laravel and contains the same user interface as the traditional Blade and Inertia stacks provided by Breeze.

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](#) and offers your choice of [Livewire](#) or [Inertia](#) driven frontend scaffolding.

Complete documentation for installing Laravel Jetstream can be found within the [official Jetstream documentation](#).

Deployment

- [Introduction](#)
- [Server Requirements](#)
- [Server Configuration](#)
 - [Nginx](#)
- [Optimization](#)
 - [Autoloader Optimization](#)
 - [Caching Configuration](#)
 - [Caching Events](#)
 - [Caching Routes](#)
 - [Caching Views](#)
- [Debug Mode](#)
- [Easy Deployment With Forge / Vapor](#)

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

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

- PHP >= 8.1
- Ctype PHP Extension
- cURL PHP Extension
- DOM PHP Extension
- Fileinfo PHP Extension
- Filter PHP Extension
- Hash PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PCRE PHP Extension
- PDO PHP Extension
- Session PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Server Configuration

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](#).**

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;
    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/php8.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
    }

    location ~ /\.(?!well-known).* {
        deny all;
    }
}

```

Optimization

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.

Caching Configuration

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`.

Caching Events

If your application is utilizing [event discovery](#), you should cache your application's event to listener mappings during your deployment process. This can be accomplished by invoking the `event:cache` Artisan command during deployment:

```
php artisan event:cache
```

Caching Routes

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.

Caching Views

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

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 application's `.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.

Easy Deployment With Forge / Vapor

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](#) 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.



Want a full guide to deploying with Laravel Forge? Check out the [Laravel Bootcamp](#) and the Forge video series available on Laracasts.

Laravel Vapor

If you would like a totally serverless, auto-scaling deployment platform tuned for Laravel, check out [Laravel Vapor](#). 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.

Chapter 3

Architecture Concepts

Request Lifecycle

- [Introduction](#)
- [Lifecycle Overview](#)
 - [First Steps](#)
 - [HTTP / Console Kernels](#)
 - [Service Providers](#)
 - [Routing](#)
 - [Finishing Up](#)
- [Focus on Service Providers](#)

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

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](#).

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](#), 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](#) that all requests must pass through before being handled by the application. These middleware handle reading and writing the [HTTP session](#), determining if the application is in maintenance mode, [verifying the CSRF token](#), 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

One of the most important kernel bootstrapping actions is loading the [service providers](#) for your application. Service providers are responsible for bootstrapping all of the framework's various components, such as the database, queue, validation, and routing components. 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.

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

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](#).

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

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

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](#)
 - [Zero Configuration Resolution](#)
 - [When to Utilize the Container](#)
- [Binding](#)
 - [Binding Basics](#)
 - [Binding Interfaces to Implementations](#)
 - [Contextual Binding](#)
 - [Binding Primitives](#)
 - [Binding Typed Variadics](#)
 - [Tagging](#)
 - [Extending Bindings](#)
- [Resolving](#)
 - [The Make Method](#)
 - [Automatic Injection](#)
- [Method Invocation and Injection](#)
- [Container Events](#)
- [PSR-11](#)

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;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Show the profile for the given user.
     */
    public function show(string $id): View
    {
        $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** 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

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 $service) {
    die($service::class);
});
```

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](#), [event listeners](#), [middleware](#), and more. Additionally, you may type-hint dependencies in the `handle` method of [queued jobs](#). Once you taste the power of automatic and zero configuration dependency injection it feels impossible to develop without it.

When to Utilize 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](#), 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](#). Secondly, if you are [writing a Laravel package](#) that you plan to share with other Laravel developers, you may need to bind your package's services into the container.

Binding

Binding Basics

Simple Bindings

Almost all of your service container bindings will be registered within [service 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;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Transistor::class, function (Application $app) {
    return new Transistor($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`:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

You may use the `bindIf` method to register a container binding only if a binding has not already been registered for the given type:

```
$this->app->bindIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```



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

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;
use Illuminate\Contracts\Foundation\Application;

$this->app->singleton(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

You may use the `singletonIf` method to register a singleton container binding only if a binding has not already been registered for the given type:

```
$this->app->singletonIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Binding Scoped Singletons

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](#) worker processes a new request or when a Laravel [queue worker](#) processes a new job:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

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

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.
 */
public function __construct(
    protected EventPusher $pusher
) {}
```

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, two controllers may depend on different implementations of the `Illuminate\Contracts\Filesystem\Filesystem` `contract`. 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

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:

```

use App\Http\Controllers\UserController;

$this->app->when(UserController::class)
    ->needs('$variableName')
    ->give($value);

```

Sometimes a class may depend on an array of tagged 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

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 filter instances.
     *
     * @var array
     */
    protected $filters;

    /**
     * Create a new class instance.
     */
    public function __construct(
        protected Logger $logger,
        Filter ...$filters,
    ) {
        $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 (Application $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

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` for the given dependency:

```
$this->app->when(ReportAggregator::class)
->needs(Report::class)
->giveTagged('reports');
```

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 (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

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 two arguments, the service class you're extending and a closure that should return the modified service. The closure receives the service being resolved and the container instance:

```
$this->app->extend(Service::class, function (Service $service, Application $app) {
    return new DecoratedService($service);
});
```

Resolving

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's 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]);
```

The `bound` method may be used to determine if a class or interface has been explicitly bound in the container:

```
if ($this->app->bound(Transistor::class)) {
    // ...
}
```

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` or the `app helper` to resolve a class instance from the container:

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

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

$transistor = app(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's constructor:

```
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 */
public function __construct(
    protected Container $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](#), [event listeners](#), [middleware](#), and more. Additionally, you may type-hint dependencies in the `handle` method of [queued jobs](#). 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;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Show the user with the given ID.
     */
    public function show(string $id): User
    {
        $user = $this->users->findOrFail($id);

        return $user;
    }
}
```

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.
     */
    public function generate(UserRepository $repository): array
    {
        return [
            // ...
        ];
    }
}
```

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

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;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor, Application $app) {
    // Called when container resolves objects of type "Transistor"...
});

$this->app->resolving(function (mixed $object, Application $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

Laravel's service container implements the [PSR-11](#) 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](#)
- [Writing Service Providers](#)
 - [The Register Method](#)
 - [The Boot Method](#)
- [Registering Providers](#)
- [Deferred 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](#).

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**. 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

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

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\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $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](#).

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\ServiceProvider;
use App\Services\DigitalOceanServiceProvider;
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 => DigitalOceanServiceProvider::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

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

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        View::composer('view', function () {
            // ...
        });
    }
}
```

Boot Method Dependency Injection

You may type-hint dependencies for your service provider's `boot` method. The [service container](#) will automatically inject any dependencies you need:

```
use Illuminate\Contracts\Routing\ResponseFactory;

/**
 * Bootstrap any application services.
 */
public function boot(ResponseFactory $response): void
{
    $response->macro('serialized', function (mixed $value) {
        // ...
    });
}
```

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 registered in this array. The default providers bootstrap the core Laravel components, such as the mailer, queue, cache, and others.

To register your provider, add it to the array:

```
'providers' => ServiceProvider::defaultProviders()->merge([
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
])->toArray(),
```

Deferred Providers

If your provider is **only** registering bindings in the [service 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\Foundation\Application;
use Illuminate\Contracts\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Get the services provided by the provider.
     *
     * @return array<int, string>
     */
    public function provides(): array
    {
        return [Connection::class];
    }
}
```

Facades

- [Introduction](#)
- [When to Utilize Facades](#)
 - [Facades vs. Dependency Injection](#)
 - [Facades vs. Helper Functions](#)
- [How Facades Work](#)
- [Real-Time Facades](#)
- [Facade Class Reference](#)

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](#). 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 - 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

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](#).

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 Utilize 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

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.
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

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

    $response->assertSee('value');
}
```

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');
});
```

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.
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

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

    $response->assertSee('value');
}
```

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;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function showProfile(string $id): View
    {
        $user = Cache::get('user:'.$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.
     */
    protected static function getFacadeAccessor(): string
    {
        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 and runs the requested method (in this case, `get`) against that object.

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.
     */
    public function publish(Publisher $publisher): void
    {
        $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 App\Contracts\Publisher;
+ use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     */
-    public function publish(Publisher $publisher): void
+    public function publish(): void
    {

        $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.
     */
    public function test_podcast_can_be_published(): void
    {
        $podcast = Podcast::factory()->create();

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

        $podcast->publish();
    }
}
```

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](#) key is also included where applicable.

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	app
Artisan	Illuminate\Contracts\Console\Kernel	artisan
Auth	Illuminate\Auth\AuthManager	auth
Auth (Instance)	Illuminate\Contracts\Auth\Guard	auth.driver
Blade	Illuminate\View\Compilers\BladeCompiler	blade.compiler
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\CacheManager	cache
Cache (Instance)	Illuminate\Cache\Repository	cache.store
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
Date	Illuminate\Support\DateFactory	date
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	db.connection
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Http	Illuminate\Http\Client\Factory	
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\LogManager	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords>PasswordBrokerManager	auth.password
Password (Instance)	Illuminate\Auth\Passwords>PasswordBroker	auth.password.broker
Pipeline (Instance)	Illuminate\Pipeline\Pipeline	
Process	Illuminate\Process\Factory	
Queue	Illuminate\Queue\QueueManager	queue
Queue (Instance)	Illuminate\Contracts\Queue\Queue	queue.connection

Queue (Base Class)	Illuminate\Queue\Queue	
RateLimiter	Illuminate\Cache\RateLimiter	
Redirect	Illuminate\Routing\Redirector	redirect
Redis	Illuminate\Redis\RedisManager	redis
Redis (Instance)	Illuminate\Redis\Connections\Connection	redis.connection
Request	Illuminate\Http\Request	request
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (Instance)	Illuminate\Http\Response	
Route	Illuminate\Routing\Router	router
Schema	Illuminate\Database\Schema\Builder	
Session	Illuminate\Session\SessionManager	session
Session (Instance)	Illuminate\Session\Store	session.store
Storage	Illuminate\Filesystem\FilesystemManager	filesystem
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	filesystem.disk
URL	Illuminate\Routing\UrlGenerator	url
Validator	Illuminate\Validation\Factory	validator
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	view
View (Instance)	Illuminate\View\View	
Vite	Illuminate\Foundation\Vite	

Chapter 4

The Basics

Routing

- [Basic Routing](#)
 - [Redirect Routes](#)
 - [View Routes](#)
 - [The Route List](#)
- [Route Parameters](#)
 - [Required Parameters](#)
 - [Optional Parameters](#)
 - [Regular Expression Constraints](#)
- [Named Routes](#)
- [Route Groups](#)
 - [Middleware](#)
 - [Controllers](#)
 - [Subdomain Routing](#)
 - [Route Prefixes](#)
 - [Route Name Prefixes](#)
- [Route Model Binding](#)
 - [Implicit Binding](#)
 - [Implicit Enum Binding](#)
 - [Explicit Binding](#)
- [Fallback Routes](#)
- [Rate Limiting](#)
 - [Defining Rate Limiters](#)
 - [Attaching Rate Limiters to Routes](#)
- [Form Method Spoofing](#)
- [Accessing the Current Route](#)
- [Cross-Origin Resource Sharing \(CORS\)](#)
- [Route Caching](#)

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

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

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 () {
    // ...
});
```



When defining multiple routes that share the same URI, routes using the `get`, `post`, `put`, `patch`, `delete`, and `options` methods should be defined before routes using the `any`, `match`, and `redirect` methods. This ensures the incoming request is matched with the correct route.

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](#). 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

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](#):

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Redirect Routes

If you are defining a route that redirects to another URL, 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

If your route only needs to return a `view`, 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`.

The Route List

The `route:list` Artisan command can easily provide an overview of all of the routes that are defined by your application:

```
php artisan route:list
```

By default, the route middleware that are assigned to each route will not be displayed in the `route:list` output; however, you can instruct Laravel to display the route middleware and middleware group names by adding the `-v` option to the command:

```
php artisan route:list -v

# Expand middleware groups...
php artisan route:list -vv
```

You may also instruct Laravel to only show routes that begin with a given URI:

```
php artisan route:list --path=api
```

In addition, you may instruct Laravel to hide any routes that are defined by third-party packages by providing the `--except-vendor` option when executing the `route:list` command:

```
php artisan route:list --except-vendor
```

Likewise, you may also instruct Laravel to only show routes that are defined by third-party packages by providing the `--only-vendor` option when executing the `route:list` command:

```
php artisan route:list --only-vendor
```

Route Parameters

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 (string $id) {
    return 'User ' . $id;
});
```

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

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string $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 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, string $id) {
    return 'User '.$id;
});
```

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 (?string $name = null) {
    return $name;
});

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

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 (string $name) {
    // ...
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function (string $id) {
    // ...
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $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 (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {
    //
})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);
```

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

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.
 */
public function boot(): void
{
    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 (string $id) {
    // Only executed if {id} is numeric...
});
```

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 (string $search) {
    return $search;
})->where('search', '.*');
```



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

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

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');

return to_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 (string $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 (string $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.

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:

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Handle an incoming request.
 *
 * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

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

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

To assign `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...
    });
});
```

Controllers

If a group of routes all utilize the same [controller](#), you may use the `controller` method to define the common controller for all of the routes within the group. Then, when defining the routes, you only need to provide the controller method that they invoke:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

Subdomain Routing

Route groups may also be used to handle subdomain routing. Subdomains may be assigned route parameters just like route URLs, 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 (string $account, string $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

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 URLs within the group with `admin`:

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Route 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 the names of all of the routes in the group 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

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

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

Typically, implicit model binding will not retrieve models that have been soft deleted. 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

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.
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

Custom Keys and 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.

If you wish, you may instruct Laravel to scope "child" bindings even when a custom key is not provided. To do so, you may invoke the `scopeBindings` method when defining your route:

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
    return $post;
})->scopeBindings();
```

Or, you may instruct an entire group of route definitions to use scoped bindings:

```
Route::scopeBindings()->group(function () {
    Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
        return $post;
    });
});
```

Similarly, you may explicitly instruct Laravel to not scope bindings by invoking the `withoutScopedBindings` method:

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
})->withoutScopedBindings();
```

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');
});
```

Implicit Enum Binding

PHP 8.1 introduced support for [Enums](#). To complement this feature, Laravel allows you to type-hint a [string-backed Enum](#) on your route definition and Laravel will only invoke the route if that route segment corresponds to a valid Enum value. Otherwise, a 404 HTTP response will be returned automatically. For example, given the following Enum:

```
<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}
```

You may define a route that will only be invoked if the `{category}` route segment is `fruits` or `people`. Otherwise, Laravel will return a 404 HTTP response:

```
use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});
```

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.
 */
public function boot(): void
{
    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

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.
 */
public function boot(): void
{
    Route::bind('user', function (string $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](#), 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

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

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, rate limiters are defined within the `boot` method of your application's `App\Providers\RouteServiceProvider` class. In fact, this class already includes a rate limiter definition that is applied to the routes in your application's `routes/api.php` file:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Define your route model bindings, pattern filters, and other route configuration.
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user() ?: $request->ip());
    });

    // ...
}
```

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\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Define your route model bindings, pattern filters, and other route configuration.
 */
protected function boot(): void
{
    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 (Request $request, array $headers) {
        return response('Custom response...', 429, $headers);
    });
});

```

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

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

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

Rate limiters may be attached to routes or route groups using the `throttle` 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

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

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 to generate the `_method` input field:

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

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](#) and [Route instance](#) to review all of the methods that are available on the router and route classes.

Cross-Origin Resource Sharing (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 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](#).

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](#)
- [Defining Middleware](#)
- [Registering Middleware](#)
 - [Global Middleware](#)
 - [Assigning Middleware to Routes](#)
 - [Middleware Groups](#)
 - [Sorting Middleware](#)
- [Middleware Parameters](#)
- [Terminable 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

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;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($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](#), so you may type-hint any dependencies you need within a middleware's constructor.

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;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // 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;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

Registering 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

If you would like to assign middleware to specific routes, you may invoke the `middleware` method when defining the route:

```
use App\Http\Middleware\Authenticate;

Route::get('/profile', function () {
    // ...
})->middleware(Authenticate::class);
```

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

```
Route::get('/', function () {
    // ...
})->middleware([First::class, Second::class]);
```

For convenience, you may assign aliases to middleware in your application's `app/Http/Kernel.php` file. By default, the `$middlewareAliases` property of this class contains entries for the middleware included with Laravel. You may add your own middleware to this list and assign it an alias of your choosing:

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

protected $middlewareAliases = [
    '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 alias has been defined in the HTTP kernel, you may use the alias when assigning middleware to routes:

```
Route::get('/profile', function () {
    // ...
})->middleware('auth');
```

Excluding Middleware

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]);
});
```

You may also exclude a given set of middleware from an entire `group` of route definitions:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

The `withoutMiddleware` method can only remove route middleware and does not apply to global 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.

Laravel includes predefined `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\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        \Illuminate\Routing\Middleware\ThrottleRequests::class.':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

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 string[]
 */
protected $middlewarePriority = [
    \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
    \Illuminate\Cookie\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
    \Illuminate\Contracts\Session\Middleware\AuthenticatesSessions::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    \Illuminate\Auth\Middleware\Authorize::class,
];
```

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;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureUserHasRole
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next, string $role): Response
    {
        if (!$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 `:`:

```
Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware('role:editor');
```

Multiple parameters may be delimited by commas:

```
Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware('role:editor,publisher');
```

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;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class TerminatingMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);
    }

    /**
     * Handle tasks after the response has been sent to the browser.
     */
    public function terminate(Request $request, Response $response): void
    {
        // ...
    }
}
```

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. 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.
 */
public function register(): void
{
    $this->app->singleton(TerminatingMiddleware::class);
}
```

CSRF Protection

- [Introduction](#)
- [Preventing CSRF Requests](#)
 - [Excluding URLs](#)
- [X-CSRF-Token](#)
- [X-XSRF-Token](#)

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](#) (CSRF) attacks.

An Explanation of the Vulnerability

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

Laravel automatically generates a CSRF "token" for each active [user session](#) managed by the application. This token is used to verify that the authenticated user is the 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, 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

If you are building a SPA that is utilizing Laravel as an API backend, you should consult the [Laravel Sanctum documentation](#) for information on authenticating with your API and protecting against CSRF vulnerabilities.

Excluding URIs From CSRF Protection

Sometimes you may wish to exclude a set of URIs from CSRF protection. For example, if you are using [Stripe](#) 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](#).

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

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](#)
- [Writing Controllers](#)
 - [Basic Controllers](#)
 - [Single Action Controllers](#)
- [Controller Middleware](#)
- [Resource Controllers](#)
 - [Partial Resource Routes](#)
 - [Nested Resources](#)
 - [Naming Resource Routes](#)
 - [Naming Resource Route Parameters](#)
 - [Scoping Resource Routes](#)
 - [Localizing Resource URLs](#)
 - [Supplementing Resource Controllers](#)
 - [Singleton Resource Controllers](#)
- [Dependency Injection and 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

Basic Controllers

To quickly generate a new controller, you may run the `make:controller` Artisan command. By default, all of the controllers for your application are stored in the `app/Http/Controllers` directory:

```
php artisan make:controller UserController
```

Let's take a look at an example of a basic controller. A controller may have any number of public methods which will respond to incoming HTTP requests:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Once you have written a controller class and method, you may define a route to the 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

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;

class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     */
    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`.

Controller Middleware

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.
     */
    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:

```
use Closure;
use Illuminate\Http\Request;

$this->middleware(function (Request $request, Closure $next) {
    return $next($request);
});
```

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 Controllers

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

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');
});
```

Soft Deleted Models

Typically, implicit model binding will not retrieve models that have been [soft deleted](#), and will instead return a 404 HTTP response. However, you can instruct the framework to allow soft deleted models by invoking the `withTrashed` method when defining your resource route:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->withTrashed();
```

Calling `withTrashed` with no arguments will allow soft deleted models for the `show`, `edit`, and `update` resource routes. You may specify a subset of these routes by passing an array to the `withTrashed` method:

```
Route::resource('photos', PhotoController::class)->withTrashed(['show']);
```

Specifying the Resource Model

If you are using [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 --model=Photo --resource
```

Generating Form Requests

You may provide the `--requests` option when generating a resource controller to instruct Artisan to generate [form request classes](#) for the controller's storage and update methods:

```
php artisan make:controller PhotoController --model=Photo --resource --requests
```

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

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

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

Laravel's [implicit model binding](#) 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](#).

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

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

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

Laravel's scoped implicit model binding 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

By default, `Route::resource` will create resource URIs using English verbs and plural rules. 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.
 */
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);

    // ...
}
```

Laravel's pluralizer supports [several different languages which you may configure based on your needs](#). Once the verbs and pluralization language have been customized, a resource route registration such as `Route::resource('publicacion', PublicacionController::class)` will produce the following URIs:

```
/publicacion/crear
/publicacion/{publicaciones}/editar
```

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.

Singleton Resource Controllers

Sometimes, your application will have resources that may only have a single instance. For example, a user's "profile" can be edited or updated, but a user may not have more than one "profile". Likewise, an image may have a single "thumbnail". These resources are called "singleton resources", meaning one and only one instance of the resource may exist. In these scenarios, you may register a "singleton" resource controller:

```
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::singleton('profile', ProfileController::class);
```

The singleton resource definition above will register the following routes. As you can see, "creation" routes are not registered for singleton resources, and the registered routes do not accept an identifier since only one instance of the resource may exist:

Verb	URI	Action	Route Name
GET	/profile	show	profile.show
GET	/profile/edit	edit	profile.edit
PUT/PATCH	/profile	update	profile.update

Singleton resources may also be nested within a standard resource:

```
Route::singleton('photos.thumbnail', ThumbnailController::class);
```

In this example, the `photos` resource would receive all of the standard resource routes; however, the `thumbnail` resource would be a singleton resource with the following routes:

Verb	URI	Action	Route Name
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update

Creatable Singleton Resources

Occasionally, you may want to define creation and storage routes for a singleton resource. To accomplish this, you may invoke the `creatable` method when registering the singleton resource route:

```
Route::singleton('photos.thumbnail', ThumbnailController::class)->creatable();
```

In this example, the following routes will be registered. As you can see, a `DELETE` route will also be registered for creatable singleton resources:

Verb	URI	Action	Route Name
GET	/photos/{photo}/thumbnail/create	create	photos.thumbnail.create
POST	/photos/{photo}/thumbnail	store	photos.thumbnail.store
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update
DELETE	/photos/{photo}/thumbnail	destroy	photos.thumbnail.destroy

If you would like Laravel to register the `DELETE` route for a singleton resource but not register the creation or storage routes, you may utilize the `destroyable` method:

```
Route::singleton(...)->destroyable();
```

API Singleton Resources

The `apiSingleton` method may be used to register a singleton resource that will be manipulated via an API, thus rendering the `create` and `edit` routes unnecessary:

```
Route::apiSingleton('profile', ProfileController::class);
```

Of course, API singleton resources may also be `creatable`, which will register `store` and `destroy` routes for the resource:

```
Route::apiSingleton('photos.thumbnail', ProfileController::class)->creatable();
```

Dependency Injection and Controllers

Constructor Injection

The Laravel [service 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
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}
}
```

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\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        // Store the user...

        return redirect('/users');
    }
}
```

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\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the given user.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Update the user...

        return redirect('/users');
    }
}
```

HTTP Requests

- [Introduction](#)
- [Interacting With The Request](#)
 - [Accessing the Request](#)
 - [Request Path, Host, and Method](#)
 - [Request Headers](#)
 - [Request IP Address](#)
 - [Content Negotiation](#)
 - [PSR-7 Requests](#)
- [Input](#)
 - [Retrieving Input](#)
 - [Input Presence](#)
 - [Merging Additional Input](#)
 - [Old Input](#)
 - [Cookies](#)
 - [Input Trimming and Normalization](#)
- [Files](#)
 - [Retrieving Uploaded Files](#)
 - [Storing Uploaded Files](#)
- [Configuring Trusted Proxies](#)
- [Configuring Trusted Hosts](#)

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

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](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->input('name');

        // Store the user...

        return redirect('/users');
    }
}
```

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 and 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\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Update the user...

        return redirect('/users');
    }
}
```

Request Path, Host, 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

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

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:

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

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']);
```

If you would like to get the current URL without a given query string parameter, you may utilize the `fullUrlWithoutQuery` method:

```
$request->fullUrlWithoutQuery(['type']);
```

Retrieving the Request Host

You may retrieve the "host" of the incoming request via the `host`, `httpHost`, and `schemeAndHttpHost` methods:

```
$request->host();
$request->httpHost();
$request->schemeAndHttpHost();
```

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

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

The `ip` method may be used to retrieve the IP address of the client that made the request to your application:

```
$ipAddress = $request->ip();
```

If you would like to retrieve an array of IP addresses, including all of the client IP addresses that were forwarded by proxies, you may use the `ips` method. The "original" client IP address will be at the end of the array:

```
$ipAddresses = $request->ips();
```

In general, IP addresses should be considered untrusted, user-controlled input and be used for informational purposes only.

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

The [PSR-7 standard](#) specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request 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

Retrieving Input

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`:

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

The `collect` method also allows you to retrieve a subset of the incoming request's input as a collection:

```
$request->collect('users')->each(function (string $user) {
    // ...
});
```

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

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

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 / objects:

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

Retrieving Stringable Input Values

Instead of retrieving the request's input data as a primitive `string`, you may use the `string` method to retrieve the request data as an instance of `Illuminate\Support\Stringable`:

```
$name = $request->string('name')->trim();
```

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 Date Input Values

For convenience, input values containing dates / times may be retrieved as Carbon instances using the `date` method. If the request does not contain an input value with the given name, `null` will be returned:

```
$birthday = $request->date('birthday');
```

The second and third arguments accepted by the `date` method may be used to specify the date's format and timezone, respectively:

```
$elapsed = $request->date('elapsed', '!H:i', 'Europe/Madrid');
```

If the input value is present but has an invalid format, an `InvalidArgumentException` will be thrown; therefore, it is recommended that you validate the input before invoking the `date` method.

Retrieving Enum Input Values

Input values that correspond to [PHP enums](#) may also be retrieved from the request. If the request does not contain an input value with the given name or the enum does not have a backing value that matches the input value, `null` will be returned. The `enum` method accepts the name of the input value and the enum class as its first and second arguments:

```
use App\Enums>Status;

$status = $request->enum('status', Status::class);
```

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

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.

Input Presence

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 `hasAny` method returns `true` if any of the specified values are present:

```
if ($request->hasAny(['name', 'email'])) {
    // ...
}
```

The `whenHas` method will execute the given closure if a value is present on the request:

```
$request->whenHas('name', function (string $input) {
    // ...
});
```

A second closure may be passed to the `whenHas` method that will be executed if the specified value is not present on the request:

```
$request->whenHas('name', function (string $input) {
    // The "name" value is present...
}, function () {
    // The "name" value is not present...
});
```

If you would like to determine if a value is present on the request and is not an empty string, you may use the `filled` method:

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

The `anyFilled` method returns `true` if any of the specified values is not an empty string:

```
if ($request->anyFilled(['name', 'email'])) {
    // ...
}
```

The `whenFilled` method will execute the given closure if a value is present on the request and is not an empty string:

```
$request->whenFilled('name', function (string $input) {
    // ...
});
```

A second closure may be passed to the `whenFilled` method that will be executed if the specified value is not "filled":

```
$request->whenFilled('name', function (string $input) {
    // The "name" value is filled...
}, function () {
    // The "name" value is not filled...
});
```

To determine if a given key is absent from the request, you may use the `missing` and `whenMissing` methods:

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

$request->whenMissing('name', function (array $input) {
    // The "name" value is missing...
}, function () {
    // The "name" value is present...
});
```

Merging Additional Input

Sometimes you may need to manually merge additional input into the request's existing input data. To accomplish this, you may use the `merge` method. If a given input key already exists on the request, it will be overwritten by the data provided to the `merge` method:

```
$request->merge(['votes' => 0]);
```

The `mergeIfMissing` method may be used to merge input into the request if the corresponding keys do not already exist within the request's input data:

```
$request->mergeIfMissing(['votes' => 0]);
```

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](#), 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

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

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

You may also use the `flashOnly` and `flashExcept` methods to flash a 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

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

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`:

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

Laravel also provides a global `old` helper. If you are displaying old input within a [Blade template](#), 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

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 and Normalization

By default, Laravel includes the `App\Http\Middleware\TrimStrings` and `Illuminate\Foundation\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.

Disabling Input Normalization

If you would like to disable this behavior for all requests, 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.

If you would like to disable string trimming and empty string conversion for a subset of requests to your application, you may use the `skipWhen` method offered by both middleware. This method accepts a closure which should return `true` or `false` to indicate if input normalization should be skipped. Typically, the `skipWhen` method should be invoked in the `boot` method of your application's `AppServiceProvider`.

```

use App\Http\Middleware\TrimStrings;
use Illuminate\Http\Request;
use Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    TrimStrings::skipWhen(function (Request $request) {
        return $request->is('admin/*');
    });

    ConvertEmptyStringsToNull::skipWhen(function (Request $request) {
        // ...
    });
}

```

Files

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

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 and 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

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

Storing Uploaded Files

To store an uploaded file, you will typically use one of your configured `filesystems`. 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](#).

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](#).

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

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<int, string>  
 */  
public function hosts(): array  
{  
    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](#)
 - [Attaching Headers to Responses](#)
 - [Attaching Cookies to Responses](#)
 - [Cookies and Encryption](#)
- [Redirects](#)
 - [Redirecting to Named Routes](#)
 - [Redirecting to Controller Actions](#)
 - [Redirecting to External Domains](#)
 - [Redirecting With Flashed Session Data](#)
- [Other Response Types](#)
 - [View Responses](#)
 - [JSON Responses](#)
 - [File Downloads](#)
 - [File Responses](#)
- [Response Macros](#)

Creating Responses

Strings and 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](#) from your routes or controllers? They will automatically be converted to JSON. Give it a shot!

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](#).

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 and Collections

You may also return [Eloquent ORM](#) 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](#):

```
use App\Models\User;

Route::get('/user/{user}', function (User $user) {
    return $user;
});
```

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

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

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` 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

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

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 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

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`, 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

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

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.
 */
public function getRouteKey(): mixed
{
    return $this->slug;
}
```

Redirecting to Controller Actions

You may also generate redirects to [controller actions](#). 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

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

Redirecting to a new URL and [flashing data to the session](#) 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`. For example, using [Blade syntax](#):

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

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 during the next request to repopulate the form:

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

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 is returned. This contract provides several helpful methods for generating responses.

View Responses

If you need control over the response's status and headers but also need to return a `view` 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

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

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

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

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 absolute 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

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, 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.
     */
    public function boot(): void
    {
        Response::macro('caps', function (string $value) {
            return Response::make(strtoupper($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](#)
 - [Writing Views in React / Vue](#)
- [Creating and Rendering Views](#)
 - [Nested View Directories](#)
 - [Creating the First Available View](#)
 - [Determining if a View Exists](#)
- [Passing Data to Views](#)
 - [Sharing Data With All Views](#)
- [View Composers](#)
 - [View Creators](#)
- [Optimizing 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. When using Laravel, view templates are usually written using the [Blade templating language](#). 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](#) to get started.

Writing Views in React / Vue

Instead of writing their frontend templates in PHP via Blade, many developers have begun to prefer to write their templates using React or Vue. Laravel makes this painless thanks to [Inertia](#), a library that makes it a cinch to tie your React / Vue frontend to your Laravel backend without the typical complexities of building an SPA.

Our Breeze and Jetstream [starter kits](#) give you a great starting point for your next Laravel application powered by Inertia. In addition, the [Laravel Bootcamp](#) provides a full demonstration of building a Laravel application powered by Inertia, including

examples in Vue and React.

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 or by using the `make:view` Artisan command:

```
php artisan make:view greeting
```

The `.blade.php` extension informs the framework that the file contains a [Blade template](#). 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](#).

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

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

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('admin.profile')) {
    // ...
}
```

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

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.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        View::share('key', 'value');
    }
}
```

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](#). 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;
use Illuminate\Support\ServiceProvider;
use Illuminate\View\View;

class ViewServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        // Using class based composers...
        Facades\View::composer('profile', ProfileComposer::class);

        // Using closure based composers...
        Facades\View::composer('welcome', function (View $view) {
            // ...
        });

        Facades\View::composer('dashboard', function (View $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
{
    /**
     * Create a new profile composer.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Bind data to the view.
     */
    public function compose(View $view): void
    {
        $view->with('count', $this->users->count());
    }
}
```

As you can see, all view composers are resolved via the [service container](#), so you may type-hint any dependencies you need within a composer's constructor.

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;
use Illuminate\Support\Facades\View;

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:

```
use Illuminate\Support\Facades;
use Illuminate\View\View;

Facades\View::composer('*', function (View $view) {
    // ...
});
```

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

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](#)
 - [Supercharging Blade With Livewire](#)
- [Displaying Data](#)
 - [HTML Entity Encoding](#)
 - [Blade and JavaScript Frameworks](#)
- [Blade Directives](#)
 - [If Statements](#)
 - [Switch Statements](#)
 - [Loops](#)
 - [The Loop Variable](#)
 - [Conditional Classes](#)
 - [Additional Attributes](#)
 - [Including Subviews](#)
 - [The `@once` Directive](#)
 - [Raw PHP](#)
 - [Comments](#)
- [Components](#)
 - [Rendering Components](#)
 - [Passing Data to Components](#)
 - [Component Attributes](#)
 - [Reserved Keywords](#)
 - [Slots](#)
 - [Inline Component Views](#)
 - [Dynamic Components](#)
 - [Manually Registering Components](#)
- [Anonymous Components](#)
 - [Anonymous Index Components](#)
 - [Data Properties / Attributes](#)
 - [Accessing Parent Data](#)
 - [Anonymous Components Paths](#)
- [Building Layouts](#)
 - [Layouts Using Components](#)
 - [Layouts Using Template Inheritance](#)
- [Forms](#)
 - [CSRF Field](#)
 - [Method Field](#)
 - [Validation Errors](#)
- [Stacks](#)
- [Service Injection](#)
- [Rendering Inline Blade Templates](#)
- [Rendering Blade Fragments](#)
- [Extending Blade](#)
 - [Custom Echo Handlers](#)
 - [Custom If Statements](#)

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 controllers using the global `view` helper. Of course, as mentioned in the documentation on [views](#), data may be passed to the Blade view using the `view` helper's second argument:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Finn']);
});
```

Supercharging Blade With Livewire

Want to take your Blade templates to the next level and build dynamic interfaces with ease? Check out [Laravel Livewire](#). Livewire allows you to write Blade components that are augmented with dynamic functionality that would typically only be possible via frontend frameworks like React or Vue, providing a great approach to building modern, reactive frontends without the complexities, client-side rendering, or build steps of many JavaScript frameworks.

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() }}.
```

HTML Entity Encoding

By default, Blade (and the Laravel `e` function) 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.
     */
    public function boot(): void
    {
        Blade::withoutDoubleEncoding();
    }
}
```

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 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 --}

@@if()

<!-- HTML output -->
@if()
```

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 `Illuminate\Support\Js::from` method directive. The `from` method accepts the same arguments as PHP's `json_encode` function; however, it will ensure that the resulting JSON is properly escaped for inclusion within HTML quotes. The `from` method will return a string `JSON.parse` JavaScript statement that will convert the given object or array into a valid JavaScript object:

```
<script>
    var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

The latest versions of the Laravel application skeleton include a `Js` facade, which provides convenient access to this functionality within your Blade templates:

```
<script>
    var app = {{ Js::from($array) }};
</script>
```



You should only use the `Js::from` method 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.

The `@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

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

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

The `@auth` and `@guest` directives may be used to quickly determine if the current user is authenticated 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

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"...
@endif
```

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
```

Session Directives

The `@session` directive may be used to determine if a `session` value exists. If the session value exists, the template contents within the `@session` and `@endsession` directives will be evaluated. Within the `@session` directive's contents, you may echo the `$value` variable to display the session value:

```
@session('status')
    <div class="p-4 bg-green-100">
        {{ $value }}
    </div>
@endsession
```

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

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
```



While iterating through a `foreach` loop, you may use 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 skip the current iteration or end the loop 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

While iterating through a `foreach` loop, 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 & Styles

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>
```

Likewise, the `@style` directive may be used to conditionally add inline CSS styles to an HTML element:

```
@php
$isActive = true;
@endphp

<span @style([
    'background-color: red',
    'font-weight: bold' => $isActive,
])></span>

<span style="background-color: red; font-weight: bold;"></span>
```

Additional Attributes

For convenience, you may use the `@checked` directive to easily indicate if a given HTML checkbox input is "checked". This directive will echo `checked` if the provided condition evaluates to `true`:

```
<input type="checkbox"
    name="active"
    value="active"
    @checked(old('active', $user->active)) />
```

Likewise, the `@selected` directive may be used to indicate if a given select option should be "selected":

```
<select name="version">
    @foreach ($product->versions as $version)
        <option value="{{ $version }}" @selected(old('version') == $version)>
            {{ $version }}
        </option>
    @endforeach
</select>
```

Additionally, the `@disabled` directive may be used to indicate if a given element should be "disabled":

```
<button type="submit" @disabled($errors->isNotEmpty())>Submit</button>
```

Moreover, the `@readonly` directive may be used to indicate if a given element should be "readonly":

```
<input type="email"
    name="email"
    value="email@laravel.com"
    @readonly($user->isNotAdmin()) />
```

In addition, the `@required` directive may be used to indicate if a given element should be "required":

```
<input type="text"
    name="title"
    value="title"
    @required($user->isAdmin()) />
```

Including Subviews



While you're free to use the `@include` directive, 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

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

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](#). For example, if you are rendering a given [component](#) 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
```

Since the `@once` directive is often used in conjunction with the `@push` or `@prepend` directives, the `@pushOnce` and `@prependOnce` directives are available for your convenience:

```
@pushOnce('scripts')
<script>
    // Your custom JavaScript...
</script>
@endPushOnce
```

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
```

Or, if you only need to use PHP to import a class, you may use the `@use` directive:

```
@use('App\Models\Flight')
```

A second argument may be provided to the `@use` directive to alias the imported class:

```
@use('App\Models\Flight', 'FlightModel')
```

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

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.

If you would like to create an anonymous component (a component with only a Blade template and no class), you may use the `--view` flag when invoking the `make:component` command:

```
php artisan make:component forms.input --view
```

The command above will create a Blade file at `resources/views/components/forms/input.blade.php` which can be rendered as a component via `<x-forms.input />`.

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(): void
{
    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.
 */
public function boot(): void
{
    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

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/>
```

If you would like to conditionally render your component, you may define a `shouldRender` method on your component class. If the `shouldRender` method returns `false` the component will not be rendered:

```
use Illuminate\Support\Str;

/**
 * Whether the component should be rendered
 */
public function shouldRender(): bool
{
    return Str::length($this->message) > 0;
}
```

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 all of the component's data attributes 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;
use Illuminate\View\View;

class Alert extends Component
{
    /**
     * Create the component instance.
     */
    public function __construct(
        public string $type,
        public string $message,
    ) {}

    /**
     * Get the view / contents that represent the component.
     */
    public function render(): View
    {
        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

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.
 */
public function __construct()
    public string $alertType,
) {}
```

The `$alertType` argument may be provided to the component like so:

```
<x-alert alert-type="danger" />
```

Short Attribute Syntax

When passing attributes to components, you may also use a "short attribute" syntax. This is often convenient since attribute names frequently match the variable names they correspond to:

```
{-- Short attribute syntax... --}
<x-profile :$userId :$name />

{-- Is equivalent to... --}
<x-profile :user-id="$userId" :name="$name" />
```

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

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.
 */
public function isSelected(string $option): bool
{
    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' : '' }} value="{{ $value }}">
{{ $label }}
</option>
```

Accessing Attributes and Slots Within Component Classes

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:

```
use Closure;

/**
 * Get the view / contents that represent the component.
 */
public function render(): Closure
{
    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

If your component requires dependencies from Laravel's [service 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.
 */
public function __construct(
    public AlertCreator $creator,
    public string $type,
    public string $message,
) {}
```

Hiding Attributes / 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 properties / methods that should not be exposed to the component template.
     *
     * @var array
     */
    protected $except = ['type'];

    /**
     * Create the component instance.
     */
    public function __construct(
        public string $type,
    ) {}
}
```

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

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

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.

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 and 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 (string $value, string $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
```

If an array is passed to the `has` method, the method will determine if all of the given attributes are present on the component:

```
@if ($attributes->has(['name', 'class']))
    <div>All of the attributes are present</div>
@endif
```

The `hasAny` method may be used to determine if any of the given attributes are present on the component:

```
@if ($attributes->hasAny(['href', ':href', 'v-bind:href']))
    <div>One of the attributes is present</div>
@endif
```

You may retrieve a specific attribute's value using the `get` method:

```
{{ $attributes->get('class') }}
```

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

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:title>
        Server Error
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

You may invoke a slot's `isEmpty` method to determine if the slot contains content:

```
<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
    @if ($slot->isEmpty())
        This is default content if the slot is empty.
    @else
        {{ $slot }}
    @endif
</div>
```

Additionally, the `hasActualContent` method may be used to determine if the slot contains any "actual" content that is not an HTML comment:

```
@if ($slot->hasActualContent())
    The scope has non-comment content.
@endif
```

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:title>
        {{ $component->formatAlert('Server Error') }}
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Slot Attributes

Like Blade components, you may assign additional [attributes](#) to slots such as CSS class names:

```
<x-card class="shadow-sm">
    <x-slot:heading class="font-bold">
        Heading
    </x-slot>

    Content

    <x-slot: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](#):

```
@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

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.
 */
public function render(): string
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    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
```

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:

```
// $componentName = "secondary-button";  
  
<x-dynamic-component :component="$componentName" class="mt-4" />
```

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.  
 */  
public function boot(): void  
{  
    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.
 */
public function boot(): void
{
    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.

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

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

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](#).

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"/>
```

Accessing Parent Data

Sometimes you may want to access data from a parent component inside a child component. In these cases, you may use the `@aware` directive. For example, imagine we are building a complex menu component consisting of a parent `<x-menu>` and child `<x-menu.item>`:

```
<x-menu color="purple">
    <x-menu.item>...</x-menu.item>
    <x-menu.item>...</x-menu.item>
</x-menu>
```

The `<x-menu>` component may have an implementation like the following:

```
<!-- /resources/views/components/menu/index.blade.php -->

@props(['color' => 'gray'])

<ul {{ $attributes->merge(['class' => 'bg-' . $color . '-200']) }}>
    {{ $slot }}
</ul>
```

Because the `color` prop was only passed into the parent (`<x-menu>`), it won't be available inside `<x-menu.item>`. However, if we use the `@aware` directive, we can make it available inside `<x-menu.item>` as well:

```
<!-- /resources/views/components/menu/item.blade.php -->

@aware(['color' => 'gray'])

<li {{ $attributes->merge(['class' => 'text-' . $color . '-800']) }}>
    {{ $slot }}
</li>
```



The `@aware` directive can not access parent data that is not explicitly passed to the parent component via HTML attributes. Default `@props` values that are not explicitly passed to the parent component can not be accessed by the `@aware` directive.

Anonymous Component Paths

As previously discussed, anonymous components are typically defined by placing a Blade template within your `resources/views/components` directory. However, you may occasionally want to register other anonymous component paths with Laravel in addition to the default path.

The `anonymousComponentPath` method accepts the "path" to the anonymous component location as its first argument and an optional "namespace" that components should be placed under as its second argument. Typically, this method should be called from the `boot` method of one of your application's [service providers](#):

```
/** 
 * Bootstrap any application services.
 */
public function boot(): void
{
    Blade::anonymousComponentPath(__DIR__.'/../components');
}
```

When component paths are registered without a specified prefix as in the example above, they may be rendered in your Blade components without a corresponding prefix as well. For example, if a `panel.blade.php` component exists in the path registered above, it may be rendered like so:

```
<x-panel />
```

Prefix "namespaces" may be provided as the second argument to the `anonymousComponentPath` method:

```
Blade::anonymousComponentPath(__DIR__.'/../components', 'dashboard');
```

When a prefix is provided, components within that "namespace" may be rendered by prefixing to the component's namespace to the component name when the component is rendered:

```
<x-dashboard::panel />
```

Building Layouts

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](#) and then use it throughout our application.

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

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](#):

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    <x-slot: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

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](#).

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

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

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 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

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

The `@error` directive may be used to quickly check 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"
       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 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

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
```

If you would like to `@push` content if a given boolean expression evaluates to `true`, you may use the `@pushIf` directive:

```
@pushIf($shouldPush, 'scripts')
<script src="/example.js"></script>
@endPushIf
```

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

The `@inject` directive may be used to retrieve a service from the Laravel [service 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>
```

Rendering Inline Blade Templates

Sometimes you may need to transform a raw Blade template string into valid HTML. You may accomplish this using the `render` method provided by the `Blade` facade. The `render` method accepts the Blade template string and an optional array of data to provide to the template:

```
use Illuminate\Support\Facades\Blade;

return Blade::render('Hello, {{ $name }}', ['name' => 'Julian Bashir']);
```

Laravel renders inline Blade templates by writing them to the `storage/framework/views` directory. If you would like Laravel to remove these temporary files after rendering the Blade template, you may provide the `deleteCachedView` argument to the method:

```
return Blade::render(
    'Hello, {{ $name }}',
    ['name' => 'Julian Bashir'],
    deleteCachedView: true
);
```

Rendering Blade Fragments

When using frontend frameworks such as [Turbo](#) and [htmx](#), you may occasionally need to only return a portion of a Blade template within your HTTP response. Blade "fragments" allow you to do just that. To get started, place a portion of your Blade template within `@fragment` and `@endfragment` directives:

```
@fragment('user-list')
<ul>
    @foreach ($users as $user)
        <li>{{ $user->name }}</li>
    @endforeach
</ul>
@endfragment
```

Then, when rendering the view that utilizes this template, you may invoke the `fragment` method to specify that only the specified fragment should be included in the outgoing HTTP response:

```
return view('dashboard', ['users' => $users])->fragment('user-list');
```

The `fragmentIf` method allows you to conditionally return a fragment of a view based on a given condition. Otherwise, the entire view will be returned:

```
return view('dashboard', ['users' => $users])
->fragmentIf($request->hasHeader('HX-Request'), 'user-list');
```

The `fragments` and `fragmentsIf` methods allow you to return multiple view fragments in the response. The fragments will be concatenated together:

```
view('dashboard', ['users' => $users])
->fragments(['user-list', 'comment-list']);

view('dashboard', ['users' => $users])
->fragmentsIf(
    $request->hasHeader('HX-Request'),
    ['user-list', 'comment-list']
);
```

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.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Blade::directive('datetime', function (string $expression) {
            return "<?php echo ($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

If you attempt to "echo" an object using Blade, the object's `__toString` method will be invoked. The `__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.
 */
public function boot(): void
{
    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

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.
 */
public function boot(): void
{
    Blade::if('disk', function (string $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
```

Asset Bundling (Vite)

- [Introduction](#)
- [Installation & Setup](#)
 - [Installing Node](#)
 - [Installing Vite and the Laravel Plugin](#)
 - [Configuring Vite](#)
 - [Loading Your Scripts and Styles](#)
- [Running Vite](#)
- [Working With JavaScript](#)
 - [Aliases](#)
 - [Vue](#)
 - [React](#)
 - [Inertia](#)
 - [URL Processing](#)
- [Working With Stylesheets](#)
- [Working With Blade and Routes](#)
 - [Processing Static Assets With Vite](#)
 - [Refreshing on Save](#)
 - [Aliases](#)
- [Custom Base URLs](#)
- [Environment Variables](#)
- [Disabling Vite in Tests](#)
- [Server-Side Rendering \(SSR\)](#)
- [Script and Style Tag Attributes](#)
 - [Content Security Policy \(CSP\) Nonce](#)
 - [Subresource Integrity \(SRI\)](#)
 - [Arbitrary Attributes](#)
- [Advanced Customization](#)
 - [Correcting Dev Server URLs](#)

Introduction

[Vite](#) is a modern frontend build tool that provides an extremely fast development environment and bundles your code for production. When building applications with Laravel, you will typically use Vite to bundle your application's CSS and JavaScript files into production ready assets.

Laravel integrates seamlessly with Vite by providing an official plugin and Blade directive to load your assets for development and production.



Are you running Laravel Mix? Vite has replaced Laravel Mix in new Laravel installations. For Mix documentation, please visit the [Laravel Mix](#) website. If you would like to switch to Vite, please see our [migration guide](#).

Choosing Between Vite and Laravel Mix

Before transitioning to Vite, new Laravel applications utilized [Mix](#), which is powered by [webpack](#), when bundling assets. Vite focuses on providing a faster and more productive experience when building rich JavaScript applications. If you are developing a Single Page Application (SPA), including those developed with tools like [Inertia](#), Vite will be the perfect fit.

Vite also works well with traditional server-side rendered applications with JavaScript "sprinkles", including those using [Livewire](#). However, it lacks some features that Laravel Mix supports, such as the ability to copy arbitrary assets into the build that are not referenced directly in your JavaScript application.

Migrating Back to Mix

Have you started a new Laravel application using our Vite scaffolding but need to move back to Laravel Mix and webpack? No problem. Please consult our [official guide on migrating from Vite to Mix](#).

Installation & Setup



The following documentation discusses how to manually install and configure the Laravel Vite plugin. However, Laravel's [starter kits](#) already include all of this scaffolding and are the fastest way to get started with Laravel and Vite.

Installing Node

You must ensure that Node.js (16+) and NPM are installed before running Vite and the Laravel plugin:

```
node -v  
npm -v
```

You can easily install the latest version of Node and NPM using simple graphical installers from [the official Node website](#). Or, if you are using [Laravel Sail](#), you may invoke Node and NPM through Sail:

```
./vendor/bin/sail node -v  
./vendor/bin/sail npm -v
```

Installing Vite and the Laravel Plugin

Within a fresh installation of Laravel, you will find a `package.json` file in the root of your application's directory structure. The default `package.json` file already includes everything you need to get started using Vite and the Laravel plugin. You may install your application's frontend dependencies via NPM:

```
npm install
```

Configuring Vite

Vite is configured via a `vite.config.js` file in the root of your project. You are free to customize this file based on your needs, and you may also install any other plugins your application requires, such as `@vitejs/plugin-vue` or `@vitejs/plugin-react`.

The Laravel Vite plugin requires you to specify the entry points for your application. These may be JavaScript or CSS files, and include preprocessed languages such as TypeScript, JSX, TSX, and Sass.

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel([
      'resources/css/app.css',
      'resources/js/app.js',
    ]),
  ],
});

```

If you are building an SPA, including applications built using Inertia, Vite works best without CSS entry points:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel([
      'resources/css/app.css',
      'resources/js/app.js',
    ]),
  ],
});

```

Instead, you should import your CSS via JavaScript. Typically, this would be done in your application's `resources/js/app.js` file:

```

import './bootstrap';
+ import '../css/app.css';

```

The Laravel plugin also supports multiple entry points and advanced configuration options such as [SSR entry points](#).

Working With a Secure Development Server

If your local development web server is serving your application via HTTPS, you may run into issues connecting to the Vite development server.

If you are using [Laravel Herd](#) and have secured the site or you are using [Laravel Valet](#) and have run the [secure command](#) against your application, the Laravel Vite plugin will automatically detect and use the generated TLS certificate for you.

If you secured the site using a host that does not match the application's directory name, you may manually specify the host in your application's `vite.config.js` file:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
+      detectTls: 'my-app.test',
    }),
  ],
});

```

When using another web server, you should generate a trusted certificate and manually configure Vite to use the generated certificates:

```

// ...
+ import fs from 'fs';

+ const host = 'my-app.test';

export default defineConfig({
  // ...
+   server: {
+     host,
+     hmr: { host },
+     https: {
+       key: fs.readFileSync(`/path/to/${host}.key`),
+       cert: fs.readFileSync(`/path/to/${host}.crt`),
+     },
+   },
});

```

If you are unable to generate a trusted certificate for your system, you may install and configure the [@vitejs/plugin-basic-ssl](#) plugin. When using untrusted certificates, you will need to accept the certificate warning for Vite's development server in your browser by following the "Local" link in your console when running the `npm run dev` command.

Running the Development Server in Sail on WSL2

When running the Vite development server within [Laravel Sail](#) on Windows Subsystem for Linux 2 (WSL2), you should add the following configuration to your `vite.config.js` file to ensure the browser can communicate with the development server:

```

// ...

export default defineConfig({
  // ...
+   server: {
+     hmr: {
+       host: 'localhost',
+     },
+   },
});

```

If your file changes are not being reflected in the browser while the development server is running, you may also need to configure Vite's `server.watch.usePolling` option.

Loading Your Scripts and Styles

With your Vite entry points configured, you may now reference them in a `@vite()` Blade directive that you add to the `<head>` of your application's root template:

```
<!doctype html>
<head>
    {{-- ... --}}

    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

If you're importing your CSS via JavaScript, you only need to include the JavaScript entry point:

```
<!doctype html>
<head>
    {{-- ... --}}

    @vite('resources/js/app.js')
</head>
```

The `@vite` directive will automatically detect the Vite development server and inject the Vite client to enable Hot Module Replacement. In build mode, the directive will load your compiled and versioned assets, including any imported CSS.

If needed, you may also specify the build path of your compiled assets when invoking the `@vite` directive:

```
<!doctype html>
<head>
    {{-- Given build path is relative to public path. --}}

    @vite('resources/js/app.js', 'vendor/courier/build')
</head>
```

Inline Assets

Sometimes it may be necessary to include the raw content of assets rather than linking to the versioned URL of the asset. For example, you may need to include asset content directly into your page when passing HTML content to a PDF generator. You may output the content of Vite assets using the `content` method provided by the `Vite` facade:

```

@php
use Illuminate\Support\Facades\Vite;
@endphp

<!doctype html>
<head>
    {{-- ... --}}


    <style>
        {!! Vite::content('resources/css/app.css') !!}
    </style>
    <script>
        {!! Vite::content('resources/js/app.js') !!}
    </script>
</head>

```

Running Vite

There are two ways you can run Vite. You may run the development server via the `dev` command, which is useful while developing locally. The development server will automatically detect changes to your files and instantly reflect them in any open browser windows.

Or, running the `build` command will version and bundle your application's assets and get them ready for you to deploy to production:

```

# Run the Vite development server...
npm run dev

# Build and version the assets for production...
npm run build

```

If you are running the development server in [Sail](#) on WSL2, you may need some [additional configuration](#) options.

Working With JavaScript

Aliases

By default, The Laravel plugin provides a common alias to help you hit the ground running and conveniently import your application's assets:

```
{
    '@' => '/resources/js'
}
```

You may overwrite the `'@'` alias by adding your own to the `vite.config.js` configuration file:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel(['resources/ts/app.tsx']),
  ],
  resolve: {
    alias: {
      '@': '/resources/ts',
    },
  },
});

```

Vue

If you would like to build your frontend using the [Vue](#) framework, then you will also need to install the `@vitejs/plugin-vue` plugin:

```
npm install --save-dev @vitejs/plugin-vue
```

You may then include the plugin in your `vite.config.js` configuration file. There are a few additional options you will need when using the Vue plugin with Laravel:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import vue from '@vitejs/plugin-vue';

export default defineConfig({
  plugins: [
    laravel(['resources/js/app.js']),
    vue({
      template: {
        transformAssetUrls: {
          // The Vue plugin will re-write asset URLs, when referenced
          // in Single File Components, to point to the Laravel web
          // server. Setting this to `null` allows the Laravel plugin
          // to instead re-write asset URLs to point to the Vite
          // server instead.
          base: null,
        },
        // The Vue plugin will parse absolute URLs and treat them
        // as absolute paths to files on disk. Setting this to
        // `false` will leave absolute URLs un-touched so they can
        // reference assets in the public directory as expected.
        includeAbsolute: false,
      },
    }),
  ],
});

```



Laravel's [starter kits](#) already include the proper Laravel, Vue, and Vite configuration. Check out [Laravel Breeze](#) for the fastest way to get started with Laravel, Vue, and Vite.

React

If you would like to build your frontend using the [React](#) framework, then you will also need to install the `@vitejs/plugin-react` plugin:

```
npm install --save-dev @vitejs/plugin-react
```

You may then include the plugin in your `vite.config.js` configuration file:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [
    laravel(['resources/js/app.jsx']),
    react(),
  ],
});
```

You will need to ensure that any files containing JSX have a `.jsx` or `.tsx` extension, remembering to update your entry point, if required, as [shown above](#).

You will also need to include the additional `@viteReactRefresh` Blade directive alongside your existing `@vite` directive.

```
@viteReactRefresh
@vite('resources/js/app.jsx')
```

The `@viteReactRefresh` directive must be called before the `@vite` directive.



Laravel's [starter kits](#) already include the proper Laravel, React, and Vite configuration. Check out [Laravel Breeze](#) for the fastest way to get started with Laravel, React, and Vite.

Inertia

The Laravel Vite plugin provides a convenient `resolvePageComponent` function to help you resolve your Inertia page components. Below is an example of the helper in use with Vue 3; however, you may also utilize the function in other frameworks such as React:

```

import { createApp, h } from 'vue';
import { createInertiaApp } from '@inertiajs/vue3';
import { resolvePageComponent } from 'laravel-vite-plugin/inertia-helpers';

createInertiaApp({
    resolve: (name) => resolvePageComponent(`./Pages/${name}.vue`,
import.meta.glob('./Pages/**/*.{vue,js}')},
    setup({ el, App, props, plugin }) {
        return createApp({ render: () => h(App, props) })
            .use(plugin)
            .mount(el)
    },
});

```



Laravel's [starter kits](#) already include the proper Laravel, Inertia, and Vite configuration. Check out [Laravel Breeze](#) for the fastest way to get started with Laravel, Inertia, and Vite.

URL Processing

When using Vite and referencing assets in your application's HTML, CSS, or JS, there are a couple of caveats to consider. First, if you reference assets with an absolute path, Vite will not include the asset in the build; therefore, you should ensure that the asset is available in your public directory.

When referencing relative asset paths, you should remember that the paths are relative to the file where they are referenced. Any assets referenced via a relative path will be re-written, versioned, and bundled by Vite.

Consider the following project structure:

```

public/
  taylor.png
resources/
  js/
    Pages/
      Welcome.vue
  images/
    abigail.png

```

The following example demonstrates how Vite will treat relative and absolute URLs:

```

<!-- This asset is not handled by Vite and will not be included in the build -->


<!-- This asset will be re-written, versioned, and bundled by Vite -->


```

Working With Stylesheets

You can learn more about Vite's CSS support within the [Vite documentation](#). If you are using PostCSS plugins such as [Tailwind](#), you may create a `postcss.config.js` file in the root of your project and Vite will automatically apply it:

```
export default {
    plugins: {
        tailwindcss: {},
        autoprefixer: {},
    },
};
```



Laravel's [starter kits](#) already include the proper Tailwind, PostCSS, and Vite configuration. Or, if you would like to use Tailwind and Laravel without using one of our starter kits, check out [Tailwind's installation guide for Laravel](#).

Working With Blade and Routes

Processing Static Assets With Vite

When referencing assets in your JavaScript or CSS, Vite automatically processes and versions them. In addition, when building Blade based applications, Vite can also process and version static assets that you reference solely in Blade templates.

However, in order to accomplish this, you need to make Vite aware of your assets by importing the static assets into the application's entry point. For example, if you want to process and version all images stored in `resources/images` and all fonts stored in `resources/fonts`, you should add the following in your application's `resources/js/app.js` entry point:

```
import.meta.glob([
    './images/**',
    './fonts/**',
]);
```

These assets will now be processed by Vite when running `npm run build`. You can then reference these assets in Blade templates using the `Vite::asset` method, which will return the versioned URL for a given asset:

```

```

Refreshing on Save

When your application is built using traditional server-side rendering with Blade, Vite can improve your development workflow by automatically refreshing the browser when you make changes to view files in your application. To get started, you can simply specify the `refresh` option as `true`.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: true,
    }),
  ],
});
```

When the `refresh` option is `true`, saving files in the following directories will trigger the browser to perform a full page refresh while you are running `npm run dev`:

- `app/View/Components/**`
- `lang/**`
- `resources/lang/**`
- `resources/views/**`
- `routes/**`

Watching the `routes/**` directory is useful if you are utilizing [Ziggy](#) to generate route links within your application's frontend.

If these default paths do not suit your needs, you can specify your own list of paths to watch:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: ['resources/views/**'],
    }),
  ],
});
```

Under the hood, the Laravel Vite plugin uses the [`vite-plugin-full-reload`](#) package, which offers some advanced configuration options to fine-tune this feature's behavior. If you need this level of customization, you may provide a `config` definition:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: [
                {
                    paths: ['path/to/watch/**'],
                    config: { delay: 300 }
                },
            ],
        }),
    ],
});

```

Aliases

It is common in JavaScript applications to [create aliases](#) to regularly referenced directories. But, you may also create aliases to use in Blade by using the `macro` method on the `\Illuminate\Support\Facades\Vite` class. Typically, "macros" should be defined within the `boot` method of a [service provider](#):

```

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Vite::macro('image', fn (string $asset) => $this->asset("resources/images/{$asset}"));
}

```

Once a macro has been defined, it can be invoked within your templates. For example, we can use the `image` macro defined above to reference an asset located at `resources/images/logo.png`:

```

```

Custom Base URLs

If your Vite compiled assets are deployed to a domain separate from your application, such as via a CDN, you must specify the `ASSET_URL` environment variable within your application's `.env` file:

```
ASSET_URL=https://cdn.example.com
```

After configuring the asset URL, all re-written URLs to your assets will be prefixed with the configured value:

```
https://cdn.example.com/build/assets/app.9dce8d17.js
```

Remember that [absolute URLs are not re-written by Vite](#), so they will not be prefixed.

Environment Variables

You may inject environment variables into your JavaScript by prefixing them with `VITE_` in your application's `.env` file:

```
VITE_SENTRY_DSN_PUBLIC=http://example.com
```

You may access injected environment variables via the `import.meta.env` object:

```
import.meta.env.VITE_SENTRY_DSN_PUBLIC
```

Disabling Vite in Tests

Laravel's Vite integration will attempt to resolve your assets while running your tests, which requires you to either run the Vite development server or build your assets.

If you would prefer to mock Vite during testing, you may call the `withoutVite` method, which is available for any tests that extend Laravel's `TestCase` class:

```
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_without_vite_example(): void
    {
        $this->withoutVite();

        // ...
    }
}
```

If you would like to disable Vite for all tests, you may call the `withoutVite` method from the `setUp` method on your base `TestCase` class:

```
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    protected function setUp(): void
    {
        parent::setUp();

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

Server-Side Rendering (SSR)

The Laravel Vite plugin makes it painless to set up server-side rendering with Vite. To get started, create an SSR entry point at `resources/js/ssr.js` and specify the entry point by passing a configuration option to the Laravel plugin:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      input: 'resources/js/app.js',
      ssr: 'resources/js/ssr.js',
    }),
  ],
});

```

To ensure you don't forget to rebuild the SSR entry point, we recommend augmenting the "build" script in your application's `package.json` to create your SSR build:

```

"scripts": {
  "dev": "vite",
  - "build": "vite build"
+ "build": "vite build && vite build --ssr"
}

```

Then, to build and start the SSR server, you may run the following commands:

```

npm run build
node bootstrap/ssr/ssr.js

```

If you are using [SSR with Inertia](#), you may instead use the `inertia:start-ssr` Artisan command to start the SSR server:

```

php artisan inertia:start-ssr

```



Laravel's [starter kits](#) already include the proper Laravel, Inertia SSR, and Vite configuration. Check out [Laravel Breeze](#) for the fastest way to get started with Laravel, Inertia SSR, and Vite.

Script and Style Tag Attributes

Content Security Policy (CSP) Nonce

If you wish to include a `nonce` attribute on your script and style tags as part of your [Content Security Policy](#), you may generate or specify a nonce using the `useCspNonce` method within a custom [middleware](#):

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Vite;
use Symfony\Component\HttpFoundation\Response;

class AddContentSecurityPolicyHeaders
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        Vite::useCspNonce();

        return $next($request)->withHeaders([
            'Content-Security-Policy' => "script-src 'nonce-".Vite::cspNonce()."''",
        ]);
    }
}
```

After invoking the `useCspNonce` method, Laravel will automatically include the `nonce` attributes on all generated script and style tags.

If you need to specify the nonce elsewhere, including the Ziggy `@route` directive included with Laravel's [starter kits](#), you may retrieve it using the `cspNonce` method:

```
@routes(nonce: Vite::cspNonce())
```

If you already have a nonce that you would like to instruct Laravel to use, you may pass the nonce to the `useCspNonce` method:

```
Vite::useCspNonce($nonce);
```

Subresource Integrity (SRI)

If your Vite manifest includes `integrity` hashes for your assets, Laravel will automatically add the `integrity` attribute on any script and style tags it generates in order to enforce [Subresource Integrity](#). By default, Vite does not include the `integrity` hash in its manifest, but you may enable it by installing the [vite-plugin-manifest-sri](#) NPM plugin:

```
npm install --save-dev vite-plugin-manifest-sri
```

You may then enable this plugin in your `vite.config.js` file:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
+ import manifestSRI from 'vite-plugin-manifest-sri';

export default defineConfig({
  plugins: [
    laravel({
      // ...
    }),
+     manifestSRI(),
  ],
});

```

If required, you may also customize the manifest key where the integrity hash can be found:

```

use Illuminate\Support\Facades\Vite;

Vite::useIntegrityKey('custom-integrity-key');

```

If you would like to disable this auto-detection completely, you may pass `false` to the `useIntegrityKey` method:

```
Vite::useIntegrityKey(false);
```

Arbitrary Attributes

If you need to include additional attributes on your script and style tags, such as the `data-turbo-track` attribute, you may specify them via the `useScriptTagAttributes` and `useStyleTagAttributes` methods. Typically, this methods should be invoked from a [service provider](#):

```

use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes([
  'data-turbo-track' => 'reload', // Specify a value for the attribute...
  'async' => true, // Specify an attribute without a value...
  'integrity' => false, // Exclude an attribute that would otherwise be included...
]);

Vite::useStyleTagAttributes([
  'data-turbo-track' => 'reload',
]);

```

If you need to conditionally add attributes, you may pass a callback that will receive the asset source path, its URL, its manifest chunk, and the entire manifest:

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes(fn (string $src, string $url, array|null $chunk, array|null
$manifest) => [
    'data-turbo-track' => $src === 'resources/js/app.js' ? 'reload' : false,
]);

Vite::useStyleTagAttributes(fn (string $src, string $url, array|null $chunk, array|null
$manifest) => [
    'data-turbo-track' => $chunk && $chunk['isEntry'] ? 'reload' : false,
]);
```



The `$chunk` and `$manifest` arguments will be `null` while the Vite development server is running.

Advanced Customization

Out of the box, Laravel's Vite plugin uses sensible conventions that should work for the majority of applications; however, sometimes you may need to customize Vite's behavior. To enable additional customization options, we offer the following methods and options which can be used in place of the `@vite` Blade directive:

```
<!doctype html>
<head>
{{-- ... --}}

{{
    Vite::useHotFile(storage_path('vite.hot')) // Customize the "hot" file...
    ->useBuildDirectory('bundle') // Customize the build directory...
    ->useManifestFilename('assets.json') // Customize the manifest filename...
    ->withEntryPoints(['resources/js/app.js']) // Specify the entry points...
    ->createAssetPathsUsing(function (string $path, ?bool $secure) { // Customize the
backend path generation for built assets...
        return "https://cdn.example.com/{$path}";
    })
}}
</head>
```

Within the `vite.config.js` file, you should then specify the same configuration:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      hotFile: 'storage/vite.hot', // Customize the "hot" file...
      buildDirectory: 'bundle', // Customize the build directory...
      input: ['resources/js/app.js'], // Specify the entry points...
    }),
  ],
  build: {
    manifest: 'assets.json', // Customize the manifest filename...
  },
});

```

Correcting Dev Server URLs

Some plugins within the Vite ecosystem assume that URLs which begin with a forward-slash will always point to the Vite dev server. However, due to the nature of the Laravel integration, this is not the case.

For example, the `vite-imagetools` plugin outputs URLs like the following while Vite is serving your assets:

```

```

The `vite-imagetools` plugin is expecting that the output URL will be intercepted by Vite and the plugin may then handle all URLs that start with `/@imagedata`. If you are using plugins that are expecting this behaviour, you will need to manually correct the URLs. You can do this in your `vite.config.js` file by using the `transformOnServe` option.

In this particular example, we will prepend the dev server URL to all occurrences of `/@imagedata` within the generated code:

```

import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import { imagedata } from 'vite-imagetools';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      transformOnServe: (code, devServerUrl) => code.replaceAll('/@imagedata',
devServerUrl+'/@imagedata'),
    }),
    imagedata(),
  ],
});

```

Now, while Vite is serving Assets, it will output URLs that point to the Vite dev server:

```

- <!-- [tl! remove] -->
+ <!-- [tl!
add] -->

```

URL Generation

- [Introduction](#)
- [The Basics](#)
 - [Generating URLs](#)
 - [Accessing the Current URL](#)
- [URLs for Named Routes](#)
 - [Signed URLs](#)
- [URLs for Controller Actions](#)
- [Default Values](#)

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

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

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`:

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

URLs for Named Routes

The `route` helper may be used to generate URLs to [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

You will often be generating URLs using the route key (typically the primary key) of [Eloquent models](#). 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

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]);
```

You may exclude the domain from the signed URL hash by providing the `absolute` argument to the `signedRoute` method:

```
return URL::signedRoute('unsubscribe', ['user' => 1], absolute: false);
```

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

To verify that an incoming request has a valid signature, you should call the `hasValidSignature` method on the incoming `Illuminate\Http\Request` instance:

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (! $request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

Sometimes, you may need to allow your application's frontend to append data to a signed URL, such as when performing client-side pagination. Therefore, you can specify request query parameters that should be ignored when validating a signed URL using the `hasValidSignatureWhileIgnoring` method. Remember, ignoring parameters allows anyone to modify those parameters on the request:

```
if (! $request->hasValidSignatureWhileIgnoring(['page', 'order'])) {
    abort(401);
}
```

Instead of validating signed URLs using the incoming request instance, you may assign the `Illuminate\Routing\Middleware\ValidateSignature` middleware to the route. If it is not already present, you may assign this middleware an alias in your HTTP kernel's `$middlewareAliases` array:

```
/**
 * The application's middleware aliases.
 *
 * Aliases may be used to conveniently assign middleware to routes and groups.
 *
 * @var array<string, class-string|string>
 */
protected $middlewareAliases = [
    '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');
```

If your signed URLs do not include the domain in the URL hash, you should provide the `relative` argument to the middleware:

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed:relative');
```

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.
 */
public function register(): void
{
    $this->renderable(function (InvalidSignatureException $e) {
        return response()->view('error.link-expired', [], 403);
    });
}
```

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

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](#) so that you have access to the current request:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;
use Symfony\Component\HttpFoundation\Response;

class SetDefaultLocaleForUrls
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        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 and Middleware Priority

Setting URL default values can interfere with Laravel's handling of implicit model bindings. Therefore, you should [prioritize your 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](#)
 - [Configuration](#)
 - [Driver Prerequisites](#)
- [Interacting With the Session](#)
 - [Retrieving Data](#)
 - [Storing Data](#)
 - [Flash Data](#)
 - [Deleting Data](#)
 - [Regenerating the Session ID](#)
- [Session Blocking](#)
- [Adding Custom Session Drivers](#)
 - [Implementing the Driver](#)
 - [Registering the Driver](#)

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](#), [Redis](#), and databases is included.

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](#) and prevents the data stored in the session from being persisted.

Driver Prerequisites

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:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('sessions', function (Blueprint $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](#):

```
php artisan session:table

php artisan migrate
```

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](#).



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

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](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function show(Request $request, string $id): View
    {
        $value = $request->session()->get('key');

        // ...

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

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

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

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](#) via the `assertSessionHas` method which is available in all of your test cases.

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();
```

Retrieving a Portion of the Session Data

The `only` and `except` methods may be used to retrieve a subset of the session data:

```
$data = $request->session()->only(['username', 'email']);
$data = $request->session()->except(['username', 'email']);
```

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 not present:

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

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

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

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

Retrieving and Deleting an Item

The `pull` method will retrieve and delete an item from the session in a single statement:

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

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

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

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

Regenerating the session ID is often done in order to prevent malicious users from exploiting a [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](#) or [Laravel 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



To utilize session blocking, your application must be using a cache driver that supports [atomic locks](#). Currently, those cache drivers include the `memcached`, `dynamodb`, `redis`, `database`, `file`, and `array` 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

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, $sessionId) {}
    public function close() {}
    public function read($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

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`. You should call the `extend` method from the `boot` method of a service provider. 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\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Session::extend('mongo', function (Application $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](#)
- [Validation Quickstart](#)
 - [Defining the Routes](#)
 - [Creating the Controller](#)
 - [Writing the Validation Logic](#)
 - [Displaying the Validation Errors](#)
 - [Repopulating Forms](#)
 - [A Note on Optional Fields](#)
 - [Validation Error Response Format](#)
- [Form Request Validation](#)
 - [Creating Form Requests](#)
 - [Authorizing Form Requests](#)
 - [Customizing the Error Messages](#)
 - [Preparing Input for Validation](#)
- [Manually Creating Validators](#)
 - [Automatic Redirection](#)
 - [Named Error Bags](#)
 - [Customizing the Error Messages](#)
 - [Performing Additional Validation](#)
- [Working With Validated Input](#)
- [Working With Error Messages](#)
 - [Specifying Custom Messages in Language Files](#)
 - [Specifying Attributes in Language Files](#)
 - [Specifying Values in Language Files](#)
- [Available Validation Rules](#)
- [Conditionally Adding Rules](#)
- [Validating Arrays](#)
 - [Validating Nested Array Input](#)
 - [Error Message Indexes and Positions](#)
- [Validating Files](#)
- [Validating Passwords](#)
- [Custom Validation Rules](#)
 - [Using Rule Objects](#)
 - [Using Closures](#)
 - [Implicit Rules](#)

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

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

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

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 Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\View\View;

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

    /**
     * Store a new blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        // Validate and store the blog post...

        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

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 `Illuminate\Validation\ValidationException` 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.
 */
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...

    return redirect('/posts');
}
```

As you can see, the validation rules are passed into the `validate` method. Don't worry - all available validation rules are [documented](#). 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](#):

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

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

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

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](#) will automatically be [flashed to the session](#).

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](#).

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

Laravel's built-in validation rules each have an error message that is located in your application's `lang/en/validation.php` file. If your application does not have a `lang` directory, you may instruct Laravel to create it using the `lang:publish` Artisan command.

Within the `lang/en/validation.php` 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 language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete [localization documentation](#).



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

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

You may use the `@error` [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](#), 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

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](#). 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](#):

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

Laravel also provides a global `old` helper. If you are displaying old input within a [Blade template](#), 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

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.

Validation Error Response Format

When your application throws a `Illuminate\Validation\ValidationException` exception and the incoming HTTP request is expecting a JSON response, Laravel will automatically format the error messages for you and return a `422 Unprocessable Entity` HTTP response.

Below, you can review an example of the JSON response format for validation errors. Note that nested error keys are flattened into "dot" notation format:

```
{
    "message": "The team name must be a string. (and 4 more errors)",
    "errors": {
        "team_name": [
            "The team name must be a string.",
            "The team name must be at least 1 characters."
        ],
        "authorization.role": [
            "The selected authorization.role is invalid."
        ],
        "users.0.email": [
            "The users.0.email field is required."
        ],
        "users.2.email": [
            "The users.2.email must be a valid email address."
        ]
    }
}
```

Form Request 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<string, \Illuminate\Contracts\Validation\Rule|array|string>
 */
public function rules(): array
{
    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](#).

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.
 */
public function store(StorePostRequest $request): RedirectResponse
{
    // 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']);

    // Store the blog post...

    return redirect('/posts');
}
```

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](#).



Need to add real-time form request validation to your Inertia powered Laravel frontend? Check out [Laravel Precognition](#).

Performing Additional Validation

Sometimes you need to perform additional validation after your initial validation is complete. You can accomplish this using the form request's `after` method.

The `after` method should return an array of callables or closures which will be invoked after validation is complete. The given callables will receive an `Illuminate\Validation\Validator` instance, allowing you to raise additional error messages if necessary:

```
use Illuminate\Validation\Validator;

/**
 * Get the "after" validation callables for the request.
 */
public function after(): array
{
    return [
        function (Validator $validator) {
            if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add(
                    'field',
                    'Something is wrong with this field!'
                );
            }
        },
    ];
}
```

As noted, the array returned by the `after` method may also contain invokable classes. The `__invoke` method of these classes will receive an `Illuminate\Validation\Validator` instance:

```

use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;
use Illuminate\Validation\Validator;

/**
 * Get the "after" validation callables for the request.
 */
public function after(): array
{
    return [
        new ValidateUserStatus,
        new ValidateShippingTime,
        function (Validator $validator) {
            //
        }
    ];
}

```

Stopping on the First Validation 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

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 `$redirectToRoute` property instead:

```

/**
 * The route that users should be redirected to if validation fails.
 *
 * @var string
 */
protected $redirectToRoute = 'dashboard';

```

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](#) within this method:

```
use App\Models\Comment;

/**
 * Determine if the user is authorized to make this request.
 */
public function authorize(): bool
{
    $comment = Comment::find($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](#), 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 remove the `authorize` method completely, or simply return `true`:

```
/*
 * Determine if the user is authorized to make this request.
 */
public function authorize(): bool
{
    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](#).

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<string, string>
 */
public function messages(): array
{
    return [
        'title.required' => 'A title is required',
        'body.required' => 'A message is required',
    ];
}
```

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<string, string>
 */
public function attributes(): array
{
    return [
        'email' => 'email address',
    ];
}
```

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.
 */
protected function prepareForValidation(): void
{
    $this->merge([
        'slug' => Str::slug($this->slug),
    ]);
}
```

Likewise, if you need to normalize any request data after validation is complete, you may use the `passedValidation` method:

```
/**
 * Handle a passed validation attempt.
 */
protected function passedValidation(): void
{
    $this->replace(['name' => 'Taylor']);
}
```

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`. The `make` method on the facade generates a new validator instance:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        $validator = Validator::make($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...

        return redirect('/posts');
    }
}
```

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

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 if validation fails:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validateWithBag('post');
```

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

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

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

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',
]);
];
```

Performing Additional Validation

Sometimes you need to perform additional validation after your initial validation is complete. You can accomplish this using the validator's `after` method. The `after` method accepts a closure or an array of callables which will be invoked after validation is complete. The given callables will receive an `Illuminate\Validation\Validator` instance, allowing you to raise additional error messages if necessary:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make(/* ... */);

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

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

As noted, the `after` method also accepts an array of callables, which is particularly convenient if your "after validation" logic is encapsulated in invokable classes, which will receive an `Illuminate\Validation\Validator` instance via their

`__invoke` method:

```
use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;

$validator->after([
    new ValidateUserStatus,
    new ValidateShippingTime,
    function ($validator) {
        // ...
    },
]);
```

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` instance, you may call the `collect` method:

```
$collection = $request->safe()->collect();
```

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

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

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

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

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

Laravel's built-in validation rules each have an error message that is located in your application's `lang/en/validation.php` file. If your application does not have a `lang` directory, you may instruct Laravel to create it using the `lang:publish` Artisan command.

Within the `lang/en/validation.php` 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 language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete [localization documentation](#).



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

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 `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

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 `lang/xx/validation.php` language file:

```
'attributes' => [
    'email' => 'email address',
],
```



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

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 `lang/xx/validation.php` language file by defining a `values` array:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

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

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

Accepted	Digits Between	JSON
Accepted If	Dimensions (Image Files)	Less Than
Active URL	Distinct	Less Than Or Equal
After (Date)	Doesnt Start With	Lowercase
After Or Equal (Date)	Doesnt End With	MAC Address
Alpha	Email	Max
Alpha Dash	Ends With	Max Digits
Alpha Numeric	Enum	MIME Types
Array	Exclude	MIME Type By File Extension
Ascii	Exclude If	Min
Bail	Exclude Unless	Min Digits
Before (Date)	Exclude With	Missing
Before Or Equal (Date)	Exclude Without	Missing If
Between	Exists (Database)	Missing Unless
Boolean	Extensions	Missing With
Confirmed	File	Missing With All
Current Password	Filled	Multiple Of
Date	Greater Than	Not In
Date Equals	Greater Than Or Equal	Not Regex
Date Format	Hex Color	Nullable
Decimal	Image (File)	Numeric
Declined	In	Present
Declined If	In Array	Present If
Different	Integer	Present Unless
Digits	IP Address	Present With

Present With All	Required Unless	Starts With
Prohibited	Required With	String
Prohibited If	Required With All	Timezone
Prohibited Unless	Required Without	Unique (Database)
Prohibits	Required Without All	Uppercase
Regular Expression	Required Array Keys	URL
Required	Same	UUID
Required If	Size	
Required If Accepted	Sometimes	

accepted

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

accepted_if:anotherfield,value,...

The field under validation must be `"yes"`, `"on"`, `1`, `"1"`, `true`, 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

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

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

The field under validation must be a value after or equal to the given date. For more information, see the [after](#) rule.

alpha

The field under validation must be entirely Unicode alphabetic characters contained in `\p{L}` and `\p{M}`.

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha:ascii',
```

alpha_dash

The field under validation must be entirely Unicode alpha-numeric characters contained in `\p{L}`, `\p{M}`, `\p{N}`, as well as ASCII dashes (`-`) and ASCII underscores (`_`).

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha_dash:ascii',
```

alpha_num

The field under validation must be entirely Unicode alpha-numeric characters contained in `\p{L}`, `\p{M}`, and `\p{N}`.

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha_num:ascii',
```

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:name,username',
]);
```

In general, you should always specify the array keys that are allowed to be present within your array.

ascii

The field under validation must be entirely 7-bit ASCII characters.

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

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` rule, the name of another field under validation may be supplied as the value of `date`.

before_or_equal:date

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` rule, the name of another field under validation may be supplied as the value of `date`.

between:min,max

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

boolean

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

confirmed

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

current_password

The field under validation must match the authenticated user's password. You may specify an [authentication guard](#) using the rule's first parameter:

```
'password' => 'current_password:api'
```

date

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

date_equals:date

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,...

The field under validation must match one of the given *formats*. 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` class.

decimal:min,max

The field under validation must be numeric and must contain the specified number of decimal places:

```
// Must have exactly two decimal places (9.99)...
'price' => 'decimal:2'

// Must have between 2 and 4 decimal places...
'price' => 'decimal:2,4'
```

declined

The field under validation must be `"no"`, `"off"`, `0`, `"0"`, `false`, or `"false"`.

declined_if:anotherfield,value,...

The field under validation must be `"no"`, `"off"`, `0`, `"0"`, `false`, or `"false"` if another field under validation is equal to a specified value.

different:*field*

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

digits:*value*

The integer under validation must have an exact length of *value*.

digits_between:*min,max*

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

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

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'
```

doesnt_start_with:*foo,bar,...*

The field under validation must not start with one of the given values.

doesn't_end_with:foo,bar,...

The field under validation must not end with one of the given values.

email

The field under validation must be formatted as an email address. This validation rule utilizes the [egulias/email-validator](#) 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`
- `filter_unicode`: `FilterEmailValidation::unicode()`

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,...

The field under validation must end with one of the given values.

enum

The `Enum` rule is a class based rule that validates whether the field under validation contains a valid enum value. The `Enum` rule accepts the name of the enum as its only constructor argument. When validating primitive values, a backed Enum should be provided to the `Enum` rule:

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rule;

$request->validate([
    'status' => [Rule::enum(ServerStatus::class)],
]);
```

The `Enum` rule's `only` and `except` methods may be used to limit which enum cases should be considered valid:

```
Rule::enum(ServerStatus::class)
    ->only([ServerStatus::Pending, ServerStatus::Active]);

Rule::enum(ServerStatus::class)
    ->except([ServerStatus::Pending, ServerStatus::Active]);
```

The `when` method may be used to conditionally modify the `Enum` rule:

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Rule;

Rule::enum(ServerStatus::class)
->when(
    Auth::user()->isAdmin(),
    fn ($rule) => $rule->only(...),
    fn ($rule) => $rule->only(...),
);
```

exclude

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods.

exclude_if:anotherfield,value

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`.

If complex conditional exclusion logic is required, you may utilize the `Rule::excludeIf` method. This method accepts a boolean or a closure. When given a closure, the closure should return `true` or `false` to indicate if the field under validation should be excluded:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf(fn () => $request->user()->is_admin),
]);
```

exclude_unless:anotherfield,value

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.

exclude_with:anotherfield

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the `anotherfield` field is present.

exclude_without:anotherfield

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the `anotherfield` field is not present.

exists:table,column

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

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

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\Database\Query\Builder;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function (Builder $query) {
            return $query->where('account_id', 1);
        }),
    ],
]);
```

You may explicitly specify the database column name that should be used by the `exists` rule generated by the `Rule::exists` method by providing the column name as the second argument to the `exists` method:

```
'state' => Rule::exists('states', 'abbreviation'),
```

`extensions:foo,bar,...`

The file under validation must have a user-assigned extension corresponding to one of the listed extensions:

```
'photo' => ['required', 'extensions:jpg,png'],
```



You should never rely on validating a file by its user-assigned extension alone. This rule should typically always be used in combination with the `mimes` or `mimetypes` rules.

file

The field under validation must be a successfully uploaded file.

filled

The field under validation must not be empty when it is present.

gt:field

The field under validation must be greater than the given *field* or *value*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](#) rule.

gte:field

The field under validation must be greater than or equal to the given *field* or *value*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](#) rule.

hex_color

The field under validation must contain a valid color value in [hexadecimal](#) format.

image

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

in:foo,bar,...

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',
    ],
    'airports.*' => Rule::in(['NYC', 'LIT']),
]);
```

in_array:*anotherfield**

The field under validation must exist in *anotherfield*'s values.

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.

ip

The field under validation must be an IP address.

ipv4

The field under validation must be an IPv4 address.

ipv6

The field under validation must be an IPv6 address.

json

The field under validation must be a valid JSON string.

lt:*field*

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` rule.

lte:*field*

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` rule.

lowercase

The field under validation must be lowercase.

mac_address

The field under validation must be a MAC address.

max:*value*

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` rule.

max_digits:*value*

The integer under validation must have a maximum length of *value*.

mimetypes:*text/plain,...*

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,...

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

```
'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>

MIME Types and Extensions

This validation rule does not verify agreement between the MIME type and the extension the user assigned to the file. For example, the `mimes:png` validation rule would consider a file containing valid PNG content to be a valid PNG image, even if the file is named `photo.txt`. If you would like to validate the user-assigned extension of the file, you may use the `extensions` rule.

min:value

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

min_digits:value

The integer under validation must have a minimum length of *value*.

multiple_of:value

The field under validation must be a multiple of *value*.

missing

The field under validation must not be present in the input data.

missing_if:anotherfield,value,...

The field under validation must not be present if the *anotherfield* field is equal to any *value*.

missing_unless:anotherfield,value

The field under validation must not be present unless the *anotherfield* field is equal to any *value*.

missing_with:foo,bar,...

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

missing_with_all:foo,bar,...

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

not_in:foo,bar,...

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

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

The field under validation may be `null`.

numeric

The field under validation must be numeric.

present

The field under validation must exist in the input data.

present_if:anotherfield,value,...

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

present_unless:anotherfield,value

The field under validation must be present unless the `anotherfield` field is equal to any `value`.

present_with:foo,bar,...

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

present_with_all:foo,bar,...

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

prohibited

The field under validation must be missing or empty. A field is "empty" if it meets one of the following criteria:

- 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 an empty path.

prohibited_if:anotherfield,value,...

The field under validation must be missing or empty if the *anotherfield* field is equal to any *value*. A field is "empty" if it meets one of the following criteria:

- 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 an empty path.

If complex conditional prohibition logic is required, you may utilize the `Rule::prohibitedIf` method. This method accepts a boolean or a closure. When given a closure, the closure should return `true` or `false` to indicate if the field under validation should be prohibited:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf(fn () => $request->user()->is_admin),
]);
```

prohibited_unless:anotherfield,value,...

The field under validation must be missing or empty unless the *anotherfield* field is equal to any *value*. A field is "empty" if it meets one of the following criteria:

- 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 an empty path.

prohibits:anotherfield,...

If the field under validation is not missing or empty, all fields in *anotherfield* must be missing or empty. A field is "empty" if it meets one of the following criteria:

- 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 an empty path.

regex:pattern

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

The field under validation must be present in the input data and not empty. A field is "empty" if it meets one of the following criteria:

- 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,...

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(fn () => $request->user()->is_admin),
]);
```

required_if_accepted:anotherfield,...

The field under validation must be present and not empty if the `anotherfield` field is equal to `"yes"`, `"on"`, `1`, `"1"`, `true`, or `"true"`.

required_unless:anotherfield,value,...

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,...

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,...

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,...

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,...

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

required_array_keys:foo,bar,...

The field under validation must be an array and must contain at least the specified keys.

same:field

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

size:value

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

The field under validation must start with one of the given values.

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

The field under validation must be a valid timezone identifier according to the `DateTimeZone::listIdentifiers` method.

The arguments accepted by the `DateTimeZone::listIdentifiers` method may also be provided to this validation rule:

```
'timezone' => 'required|timezone:all';

'timezone' => 'required|timezone:Africa';

'timezone' => 'required|timezone:per_country,US';
```

unique:table,column

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(fn (Builder $query) => $query->where('account_id', 1))
```

uppercase

The field under validation must be uppercase.

url

The field under validation must be a valid URL.

If you would like to specify the URL protocols that should be considered valid, you may pass the protocols as validation rule parameters:

```
'url' => 'url:http,https',
'game' => 'url:minecraft,steam',
```

ulid

The field under validation must be a valid [Universally Unique Lexicographically Sortable Identifier](#) (ULID).

uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

Conditionally Adding Rules

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

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](#).

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.

```
use Illuminate\Support\Fluent;

$validator->sometimes('reason', 'required|max:500', function (Fluent $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 (Fluent $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

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 (Fluent $input, Fluent $item) {
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function (Fluent $input, Fluent $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

As discussed in the [array validation rule documentation](#), 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:name,username',
]);
```

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.

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](#), 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',
    ]
],
```

Accessing Nested Array Data

Sometimes you may need to access the value for a given nested array element when assigning validation rules to the attribute. You may accomplish this using the `Rule::forEach` method. The `forEach` method accepts a closure that will be invoked for each iteration of the array attribute under validation and will receive the attribute's value and explicit, fully-expanded attribute name. The closure should return an array of rules to assign to the array element:

```
use App\Rules\HasPermission;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$validator = Validator::make($request->all(), [
    'companies.*.id' => Rule::forEach(function (string|null $value, string $attribute) {
        return [
            Rule::exists(Company::class, 'id'),
            new HasPermission('manage-company', $value),
        ];
    }),
]);
```

Error Message Indexes and Positions

When validating arrays, you may want to reference the index or position of a particular item that failed validation within the error message displayed by your application. To accomplish this, you may include the `:index` (starts from `0`) and `:position` (starts from `1`) placeholders within your [custom validation message](#):

```
use Illuminate\Support\Facades\Validator;

$input = [
    'photos' => [
        [
            'name' => 'BeachVacation.jpg',
            'description' => 'A photo of my beach vacation!',
        ],
        [
            'name' => 'GrandCanyon.jpg',
            'description' => '',
        ],
    ],
];

Validator::validate($input, [
    'photos.*.description' => 'required',
], [
    'photos.*.description.required' => 'Please describe photo #:position.',
]);

```

Given the example above, validation will fail and the user will be presented with the following error of *"Please describe photo #2."*

If necessary, you may reference more deeply nested indexes and positions via `second-index`, `second-position`, `third-index`, `third-position`, etc.

```
'photos.*.attributes.*.string' => 'Invalid attribute for photo #:second-position.',
```

Validating Files

Laravel provides a variety of validation rules that may be used to validate uploaded files, such as `mimes`, `image`, `min`, and `max`. While you are free to specify these rules individually when validating files, Laravel also offers a fluent file validation rule builder that you may find convenient:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'attachment' => [
        'required',
        File::types(['mp3', 'wav'])
            ->min(1024)
            ->max(12 * 1024),
    ],
]);

```

If your application accepts images uploaded by your users, you may use the `File` rule's `image` constructor method to indicate that the uploaded file should be an image. In addition, the `dimensions` rule may be used to limit the dimensions of the image:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'photo' => [
        'required',
        File::image()
            ->min(1024)
            ->max(12 * 1024)
            ->dimensions(Rule::dimensions() ->maxWidth(1000) ->maxHeight(500)),
    ],
]);
```



More information regarding validating image dimensions may be found in the [dimension rule documentation](#).

File Sizes

For convenience, minimum and maximum file sizes may be specified as a string with a suffix indicating the file size units. The `kb`, `mb`, `gb`, and `tb` suffixes are supported:

```
File::image()
    ->min('1kb')
    ->max('10mb')
```

File Types

Even though you only need to specify the extensions when invoking the `types` method, this method 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>

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](#) model to determine if a password has been leaked via the [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

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.
 */
public function boot(): void
{
    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()],
```

Occasionally, you may want to attach additional validation rules to your default password validation rules. You may use the `rules` method to accomplish this:

```
use App\Rules\ZxcvbnRule;

Password::defaults(function () {
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

Custom Validation Rules

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 a single method: `validate`. This method receives the attribute name, its value, and a callback that should be invoked on failure with the validation error message:

```
<?php

namespace App\Rules;

use Closure;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements ValidationRule
{
    /**
     * Run the validation rule.
     */
    public function validate(string $attribute, mixed $value, Closure $fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail('The :attribute must be 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],
]);
```

Translating Validation Messages

Instead of providing a literal error message to the `$fail` closure, you may also provide a [translation string key](#) and instruct Laravel to translate the error message:

```
if (strtoupper($value) !== $value) {
    $fail('validation.uppercase')->translate();
}
```

If necessary, you may provide placeholder replacements and the preferred language as the first and second arguments to the `translate` method:

```
$fail('validation.location')->translate([
    'value' => $this->value,
], 'fr')
```

Accessing Additional Data

If your custom validation rule class needs to access all of the other data undergoing validation, your rule class may implement the [Illuminate\Contracts\Validation\DataAwareRule](#) interface. This interface requires your class to define a `setData` method. This method will automatically be invoked by Laravel (before validation proceeds) with all of the data under validation:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements DataAwareRule, ValidationRule
{
    /**
     * All of the data under validation.
     *
     * @var array<string, mixed>
     */
    protected $data = [];

    // ...

    /**
     * Set the data under validation.
     *
     * @param array<string, mixed> $data
     */
    public function setData(array $data): static
    {
        $this->data = $data;

        return $this;
    }
}
```

Or, if your validation rule requires access to the validator instance performing the validation, you may implement the `ValidatorAwareRule` interface:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
use Illuminate\Validation\Validator;

class Uppercase implements ValidationRule, ValidatorAwareRule
{
    /**
     * The validator instance.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * Set the current validator.
     */
    public function setValidator(Validator $validator): static
    {
        $this->validator = $validator;

        return $this;
    }
}
```

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;
use Closure;

$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function (string $attribute, mixed $value, Closure $fail) {
            if ($value === 'foo') {
                $fail("The {$attribute} is invalid.");
            }
        },
    ],
]);
```

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` 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 quickly generate a new implicit rule object, you may use the `make:rule` Artisan command with the `--implicit` option:

```
php artisan make:rule Uppercase --implicit
```



An "implicit" rule only implies that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Error Handling

- [Introduction](#)
- [Configuration](#)
- [The Exception Handler](#)
 - [Reporting Exceptions](#)
 - [Exception Log Levels](#)
 - [Ignoring Exceptions by Type](#)
 - [Rendering Exceptions](#)
 - [Reportable and Renderable Exceptions](#)
- [Throttling Reported Exceptions](#)
- [HTTP Exceptions](#)
 - [Custom HTTP Error Pages](#)

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

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

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](#), [Bugsnag](#), or [Sentry](#). By default, exceptions will be logged based on your `logging` configuration. However, you are free to log exceptions however you wish.

If you need to report different types of exceptions in different ways, you may use the `reportable` method to register a closure that should be executed when an exception of a given type needs to be reported. Laravel will determine 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.
 */
public function register(): void
{
    $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](#).

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 defining a `context` method on 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<string, mixed>
 */
protected function context(): array
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

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 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<string, mixed>
     */
    public function context(): array
    {
        return ['order_id' => $this->orderId];
    }
}
```

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(string $value): bool
{
    try {
        // Validate the value...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
}
```

Deduplicating Reported Exceptions

If you are using the `report` function throughout your application, you may occasionally report the same exception multiple times, creating duplicate entries in your logs.

If you would like to ensure that a single instance of an exception is only ever reported once, you may set the `$withoutDuplicates` property to `true` within your application's `App\Exceptions\Handler` class:

```

namespace App\Exceptions;

use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    /**
     * Indicates that an exception instance should only be reported once.
     *
     * @var bool
     */
    protected $withoutDuplicates = true;

    // ...
}

```

Now, when the `report` helper is called with the same instance of an exception, only the first call will be reported:

```

$original = new RuntimeException('Whoops!');

report($original); // reported

try {
    throw $original;
} catch (Throwable $caught) {
    report($caught); // ignored
}

report($original); // ignored
report($caught); // ignored

```

Exception Log Levels

When messages are written to your application's `logs`, the messages are written at a specified `log level`, which indicates the severity or importance of the message being logged.

As noted above, even 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; however, since the log level can sometimes influence the channels on which a message is logged, you may wish to configure the log level that certain exceptions are logged at.

To accomplish this, you may define a `$levels` property on your application's exception handler. This property should contain an array of exception types and their associated log levels:

```
use PDOException;
use Psr\Log\LogLevel;

/**
 * A list of exception types with their corresponding custom log levels.
 *
 * @var array<class-string<\Throwable>, \Psr\Log\LogLevel::*>
 */
protected $levels = [
    PDOException::class => LogLevel::CRITICAL,
];
```

Ignoring Exceptions by Type

When building your application, there will be some types of exceptions you never want to report. To ignore these exceptions, define a `$dontReport` property on your application's exception handler. 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 are not reported.
 *
 * @var array<int, class-string<\Throwable>>
 */
protected $dontReport = [
    InvalidOrderException::class,
];
```

Internally, Laravel already ignores some types of errors for you, such as exceptions resulting from 404 HTTP errors or 419 HTTP responses generated by invalid CSRF tokens. If you would like to instruct Laravel to stop ignoring a given type of exception, you may invoke the `stopIgnoring` method within your exception handler's `register` method:

```
use Symfony\Component\HttpKernel\Exception\HttpException;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->stopIgnoring(HttpException::class);

    // ...
}
```

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 by invoking the `renderable` method within 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 determine what type of exception the closure renders by examining the type-hint of the closure:

```

use App\Exceptions\InvalidOrderException;
use Illuminate\Http\Request;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->renderable(function (InvalidOrderException $e, Request $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 Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->renderable(function (NotFoundHttpException $e, Request $request) {
        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Record not found.'
            ], 404);
        }
    });
}

```

Reportable and Renderable Exceptions

Instead of defining custom reporting and rendering behavior in your exception handler's `register` method, you may define `report` and `render` methods directly on your application's exceptions. When these methods exist, they will automatically be called by the framework:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Http\Request;
use Illuminate\Http\Response;

class InvalidOrderException extends Exception
{
    /**
     * Report the exception.
     */
    public function report(): void
    {
        // ...
    }

    /**
     * Render the exception into an HTTP response.
     */
    public function render(Request $request): Response
    {
        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.
 */
public function render(Request $request): Response|bool
{
    if (/* Determine if the exception needs custom rendering */) {

        return response(/* ... */);
    }

    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.
 */
public function report(): bool
{
    if (/** Determine if the exception needs custom reporting */) {

        // ...

        return true;
    }

    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](#).

Throttling Reported Exceptions

If your application reports a very large number of exceptions, you may want to throttle how many exceptions are actually logged or sent to your application's external error tracking service.

To take a random sample rate of exceptions, you can return a `Lottery` instance from your exception handler's `throttle` method. If your `App\Exceptions\Handler` class does not contain this method, you may simply add it to the class:

```
use Illuminate\Support\Lottery;
use Throwable;

/**
 * Throttle incoming exceptions.
 */
protected function throttle(Throwable $e): mixed
{
    return Lottery::odds(1, 1000);
}
```

It is also possible to conditionally sample based on the exception type. If you would like to only sample instances of a specific exception class, you may return a `Lottery` instance only for that class:

```

use App\Exceptions\ApiMonitoringException;
use Illuminate\Support\Lottery;
use Throwable;

/**
 * Throttle incoming exceptions.
 */
protected function throttle(Throwable $e): mixed
{
    if ($e instanceof ApiMonitoringException) {
        return Lottery::odds(1, 1000);
    }
}

```

You may also rate limit exceptions logged or sent to an external error tracking service by returning a `Limit` instance instead of a `Lottery`. This is useful if you want to protect against sudden bursts of exceptions flooding your logs, for example, when a third-party service used by your application is down:

```

use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;
use Throwable;

/**
 * Throttle incoming exceptions.
 */
protected function throttle(Throwable $e): mixed
{
    if ($e instanceof BroadcastException) {
        return Limit::perMinute(300);
    }
}

```

By default, limits will use the exception's class as the rate limit key. You can customize this by specifying your own key using the `by` method on the `Limit`:

```

use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;
use Throwable;

/**
 * Throttle incoming exceptions.
 */
protected function throttle(Throwable $e): mixed
{
    if ($e instanceof BroadcastException) {
        return Limit::perMinute(300)-->by($e->getMessage());
    }
}

```

Of course, you may return a mixture of `Lottery` and `Limit` instances for different exceptions:

```

use App\Exceptions\ApiMonitoringException;
use Illuminate\\Broadcasting\\BroadcastException;
use Illuminate\\Cache\\RateLimiting\\Limit;
use Illuminate\\Support\\Lottery;
use Throwable;

/**
 * Throttle incoming exceptions.
 */
protected function throttle(Throwable $e): mixed
{
    return match (true) {
        $e instanceof BroadcastException => Limit::perMinute(300),
        $e instanceof ApiMonitoringException => Lottery::odds(1, 1000),
        default => Limit::none(),
    };
}

```

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

Laravel makes it easy to display custom error pages for various HTTP status codes. For example, 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 for 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\ErrorException` 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
```

Fallback HTTP Error Pages

You may also define a "fallback" error page for a given series of HTTP status codes. This page will be rendered if there is not a corresponding page for the specific HTTP status code that occurred. To accomplish this, define a `4xx.blade.php` template and a `5xx.blade.php` template in your application's `resources/views/errors` directory.

Logging

- [Introduction](#)
- [Configuration](#)
 - [Available Channel Drivers](#)
 - [Channel Prerequisites](#)
 - [Logging Deprecation Warnings](#)
- [Building Log Stacks](#)
- [Writing Log Messages](#)
 - [Contextual Information](#)
 - [Writing to Specific Channels](#)
- [Monolog Channel Customization](#)
 - [Customizing Monolog for Channels](#)
 - [Creating Monolog Handler Channels](#)
 - [Creating Custom Channels via Factories](#)
- [Tailing Log Messages Using Pail](#)
 - [Installation](#)
 - [Usage](#)
 - [Filtering Logs](#)

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](#) 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

All of the configuration options for your application's logging behavior are 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](#).

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

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 <code>RotatingFileHandler</code> based Monolog driver which rotates daily
errorlog	An <code>ErrorLogHandler</code> based Monolog driver
monolog	A Monolog factory driver that may use any supported Monolog handler
papertrail	A <code>SyslogUdpHandler</code> based Monolog driver
single	A single file or path based logger channel (<code>StreamHandler</code>)
slack	A <code>SlackWebhookHandler</code> based Monolog driver
stack	A wrapper to facilitate creating "multi-channel" channels
syslog	A <code>SyslogHandler</code> based Monolog driver



Check out the documentation on [advanced channel customization](#) to learn more about the `monolog` and `custom` drivers.

Channel Prerequisites

Configuring the Single and Daily Channels

The `single` and `daily` channels have three optional configuration options: `bubble`, `permission`, and `locking`.

Name	Description	Default
<code>bubble</code>	Indicates if messages should bubble up to other channels after being handled	<code>true</code>
<code>locking</code>	Attempt to lock the log file before writing to it	<code>false</code>
<code>permission</code>	The log file's permissions	<code>0644</code>

Additionally, the retention policy for the `daily` channel can be configured via the `days` option:

Name	Description	Default
<code>days</code>	The number of days that daily log files should be retained	7

Configuring the Papertrail Channel

The `papertrail` channel requires the `host` and `port` configuration options. You can obtain these values from [Papertrail](#).

Configuring the Slack Channel

The `slack` channel requires a `url` configuration option. This URL should match a URL for an [incoming webhook](#) 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.

Logging Deprecation Warnings

PHP, Laravel, and other libraries often notify their users that some of their features have been deprecated and will be removed in a future version. If you would like to log these deprecation warnings, you may specify your preferred `deprecations` log channel in your application's `config/logging.php` configuration file:

```
'deprecations' => env('LOG_DEPRECATED_CHANNEL', 'null'),  
  
'channels' => [  
    ...  
]
```

Or, you may define a log channel named `deprecations`. If a log channel with this name exists, it will always be used to log deprecations:

```
'channels' => [  
    'deprecations' => [  
        'driver' => 'single',  
        'path' => storage_path('logs/php-deprecation-warnings.log'),  
    ],  
],
```

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

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](#). In descending order of severity, these log levels are: `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

You may write information to the logs using the `Log facade`. As previously mentioned, the logger provides the eight logging levels defined in the [RFC 5424 specification](#): `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;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function show(string $id): View
    {
        Log::info('Showing the user profile for user: {id}', ['id' => $id]);

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

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 {id} failed to login.', ['id' => $user->id]);
```

Occasionally, you may wish to specify some contextual information that should be included with all subsequent log entries in a particular channel. 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\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::withContext([
            'request-id' => $requestId
        ]);

        $response = $next($request);

        $response->headers->set('Request-Id', $requestId);

        return $response;
    }
}
```

If you would like to share contextual information across *all* logging channels, you may invoke the `Log::shareContext()` method. This method will provide the contextual information to all created channels and any channels that are created subsequently:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::shareContext([
            'request-id' => $requestId
        ]);

        // ...
    }
}
```



If you need to share log context while processing queued jobs, you may utilize [job middleware](#).

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!');
```

On-Demand Channels

It is also possible to create an on-demand channel by providing the configuration at runtime without that configuration being present in your application's `logging` configuration file. To accomplish this, you may pass a configuration array to the `Log` facade's `build` method:

```
use Illuminate\Support\Facades\Log;

Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
])->info('Something happened!');
```

You may also wish to include an on-demand channel in an on-demand logging stack. This can be achieved by including your on-demand channel instance in the array passed to the `stack` method:

```
use Illuminate\Support\Facades\Log;

$channel = Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
]);

Log::stack(['slack', $channel])->info('Something happened!');
```

Monolog Channel Customization

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 Illuminate\Log\Logger;
use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    /**
     * Customize the given logger instance.
     */
    public function __invoke(Logger $logger): void
    {
        foreach ($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](#), so any constructor dependencies they require will automatically be injected.

Creating Monolog Handler Channels

Monolog has a variety of [available handlers](#) 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

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',
],
],
```

Monolog Processors

Monolog can also process messages before logging them. You can create your own processors or use the [existing processors offered by Monolog](#).

If you would like to customize the processors for a `monolog` driver, add a `processors` configuration value to your channel's configuration:

```
'memory' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\StreamHandler::class,
    'with' => [
        'stream' => 'php://stderr',
    ],
    'processors' => [
        // Simple syntax...
        Monolog\Processor\MemoryUsageProcessor::class,
        // With options...
        [
            'processor' => Monolog\Processor\PsrLogMessageProcessor::class,
            'with' => ['removeUsedContextFields' => true],
        ],
    ],
],
```

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.
     */
    public function __invoke(array $config): Logger
    {
        return new Logger(/* ... */);
    }
}
```

Tailing Log Messages Using Pail

Often you may need to tail your application's logs in real time. For example, when debugging an issue or when monitoring your application's logs for specific types of errors.

Laravel Pail is a package that allows you to easily dive into your Laravel application's log files directly from the command line. Unlike the standard `tail` command, Pail is designed to work with any log driver, including Sentry or Flare. In addition, Pail provides a set of useful filters to help you quickly find what you're looking for.

Installation



Laravel Pail requires PHP 8.2+ and the PCNTL extension.

To get started, install Pail into your project using the Composer package manager:

```
composer require laravel/pail
```

Usage

To start tailing logs, run the `pail` command:

php artisan pail

To increase the verbosity of the output and avoid truncation (...), use the `-v` option:

```
php artisan pail -v
```

For maximum verbosity and to display exception stack traces, use the `-vv` option:

```
php artisan tail -vv
```

To stop tailing logs, press `Ctrl+C` at any time.

Filtering Logs

--filter

You may use the `--filter` option to filter logs by their type, file, message, and stack trace content:

```
php artisan tail --filter="QueryException"
```

--message

To filter logs by only their message, you may use the `--message` option:

```
php artisan tail --message="User created"
```

--level

The `--level` option may be used to filter logs by their log level:

```
php artisan tail --level=error
```

--user

To only display logs that were written while a given user was authenticated, you may provide the user's ID to the `--user` option:

```
php artisan tail --user=1
```

Chapter 5

Digging Deeper

Artisan Console

- [Introduction](#)
 - [Tinker \(REPL\)](#).
- [Writing Commands](#)
 - [Generating Commands](#)
 - [Command Structure](#)
 - [Closure Commands](#)
 - [Isolatable Commands](#)
- [Defining Input Expectations](#)
 - [Arguments](#)
 - [Options](#)
 - [Input Arrays](#)
 - [Input Descriptions](#)
 - [Prompting for Missing Input](#)
- [Command I/O](#)
 - [Retrieving Input](#)
 - [Prompting for Input](#)
 - [Writing Output](#)
- [Registering Commands](#)
- [Programmatically Executing Commands](#)
 - [Calling Commands From Other Commands](#)
- [Signal Handling](#)
- [Stub Customization](#)
- [Events](#)

Introduction

Artisan is the command line interface included with Laravel. Artisan exists at the root of your application as the `artisan` script and provides a number of helpful commands that can assist you while you build your application. To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

Every command also includes a "help" screen which displays and describes the command's available arguments and options. To view a help screen, precede the name of the command with `help`:

```
php artisan help migrate
```

Laravel Sail

If you are using [Laravel Sail](#) as your local development environment, remember to use the `sail` command line to invoke Artisan commands. Sail will execute your Artisan commands within your application's Docker containers:

```
./vendor/bin/sail artisan list
```

Tinker (REPL)

Laravel Tinker is a powerful REPL for the Laravel framework, powered by the [PsySH](#) package.

Installation

All Laravel applications include Tinker by default. However, you may install Tinker using Composer if you have previously removed it from your application:

```
composer require laravel/tinker
```



Looking for hot reloading, multiline code editing, and autocompletion when interacting with your Laravel application? Check out [Tinkerwell!](#)

Usage

Tinker allows you to interact with your entire Laravel application on the command line, including your Eloquent models, jobs, events, and more. To enter the Tinker environment, run the `tinker` Artisan command:

```
php artisan tinker
```

You can publish Tinker's configuration file using the `vendor:publish` command:

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```



The `dispatch` helper function and `dispatch` method on the `Dispatchable` class depends on garbage collection to place the job on the queue. Therefore, when using tinker, you should use `Bus::dispatch` or `Queue::push` to dispatch jobs.

Command Allow List

Tinker utilizes an "allow" list to determine which Artisan commands are allowed to be run within its shell. By default, you may run the `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize`, and `up` commands. If you would like to allow more commands you may add them to the `commands` array in your `tinker.php` configuration file:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

Classes That Should Not Be Aliased

Typically, Tinker automatically aliases classes as you interact with them in Tinker. However, you may wish to never alias some classes. You may accomplish this by listing the classes in the `dont_alias` array of your `tinker.php` configuration file:

```
'dont_alias' => [
    App\Models\User::class,
],
```

Writing Commands

In addition to the commands provided with Artisan, you may build your own custom commands. Commands are typically stored in the `app\Console\Commands` directory; however, you are free to choose your own storage location as long as

your commands can be loaded by Composer.

Generating Commands

To create a new command, you may use the `make:command` Artisan command. This command will create a new command class in the `app\Console\Commands` directory. Don't worry if this directory does not exist in your application - it will be created the first time you run the `make:command` Artisan command:

```
php artisan make:command SendEmails
```

Command Structure

After generating your command, you should define appropriate values for the `signature` and `description` properties of the class. These properties will be used when displaying your command on the `list` screen. The `signature` property also allows you to define [your command's input expectations](#). The `handle` method will be called when your command is executed. You may place your command logic in this method.

Let's take a look at an example command. Note that we are able to request any dependencies we need via the command's `handle` method. The Laravel [service container](#) will automatically inject all dependencies that are type-hinted in this method's signature:

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use App\Support\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send a marketing email to a user';

    /**
     * Execute the console command.
     */
    public function handle(DripEmailer $drip): void
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```



For greater code reuse, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks. In the example above, note that we inject a service class to do the "heavy lifting" of sending the e-mails.

Closure Commands

Closure based commands provide an alternative to defining console commands as classes. In the same way that route closures are an alternative to controllers, think of command closures as an alternative to command classes. Within the `commands` method of your `app\Console\Kernel.php` file, Laravel loads the `routes/console.php` file:

```
/**
 * Register the closure based commands for the application.
 */
protected function commands(): void
{
    require base_path('routes/console.php');
}
```

Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application. Within this file, you may define all of your closure based console commands using the `Artisan::command` method. The `command` method accepts two arguments: the `command signature` and a closure which receives the command's arguments and options:

```
Artisan::command('mail:send {user}', function (string $user) {
    $this->info("Sending email to: {$user}!");
});
```

The closure is bound to the underlying command instance, so you have full access to all of the helper methods you would typically be able to access on a full command class.

Type-Hinting Dependencies

In addition to receiving your command's arguments and options, command closures may also type-hint additional dependencies that you would like resolved out of the `service container`:

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, string $user) {
    $drip->send(User::find($user));
});
```

Closure Command Descriptions

When defining a closure based command, you may use the `purpose` method to add a description to the command. This description will be displayed when you run the `php artisan list` or `php artisan help` commands:

```
Artisan::command('mail:send {user}', function (string $user) {
    // ...
})->purpose('Send a marketing email to a user');
```

Isolatable Commands



To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`, or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Sometimes you may wish to ensure that only one instance of a command can run at a time. To accomplish this, you may implement the `Illuminate\Contracts\Console\Isolatable` interface on your command class:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\Isolatable;

class SendEmails extends Command implements Isolatable
{
    // ...
}
```

When a command is marked as `Isolatable`, Laravel will automatically add an `--isolated` option to the command. When the command is invoked with that option, Laravel will ensure that no other instances of that command are already running. Laravel accomplishes this by attempting to acquire an atomic lock using your application's default cache driver. If other instances of the command are running, the command will not execute; however, the command will still exit with a successful exit status code:

```
php artisan mail:send 1 --isolated
```

If you would like to specify the exit status code that the command should return if it is not able to execute, you may provide the desired status code via the `isolated` option:

```
php artisan mail:send 1 --isolated=12
```

Lock ID

By default, Laravel will use the command's name to generate the string key that is used to acquire the atomic lock in your application's cache. However, you may customize this key by defining an `isolatableId` method on your Artisan command class, allowing you to integrate the command's arguments or options into the key:

```
/**
 * Get the isolatable ID for the command.
 */
public function isolatableId(): string
{
    return $this->argument('user');
}
```

Lock Expiration Time

By default, isolation locks expire after the command is finished. Or, if the command is interrupted and unable to finish, the lock will expire after one hour. However, you may adjust the lock expiration time by defining a `isolationLockExpiresAt` method on your command:

```
use DateTimeInterface;
use DateInterval;

/**
 * Determine when an isolation lock expires for the command.
 */
public function isolationLockExpiresAt(): DateTimeInterface|DateInterval
{
    return now()->addMinutes(5);
}
```

Defining Input Expectations

When writing console commands, it is common to gather input from the user through arguments or options. Laravel makes it very convenient to define the input you expect from the user using the `signature` property on your commands. The `signature` property allows you to define the name, arguments, and options for the command in a single, expressive, route-like syntax.

Arguments

All user supplied arguments and options are wrapped in curly braces. In the following example, the command defines one required argument: `user`:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user}';
```

You may also make arguments optional or define default values for arguments:

```
// Optional argument...
'mail:send {user?}'
```



```
// Optional argument with default value...
'mail:send {user=foo}'
```

Options

Options, like arguments, are another form of user input. Options are prefixed by two hyphens (`--`) when they are provided via the command line. There are two types of options: those that receive a value and those that don't. Options that don't receive a value serve as a boolean "switch". Let's take a look at an example of this type of option:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

In this example, the `--queue` switch may be specified when calling the Artisan command. If the `--queue` switch is passed, the value of the option will be `true`. Otherwise, the value will be `false`:

```
php artisan mail:send 1 --queue
```

Options With Values

Next, let's take a look at an option that expects a value. If the user must specify a value for an option, you should suffix the option name with a `=` sign:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue=}';
```

In this example, the user may pass a value for the option like so. If the option is not specified when invoking the command, its value will be `null`:

```
php artisan mail:send 1 --queue=default
```

You may assign default values to options by specifying the default value after the option name. If no option value is passed by the user, the default value will be used:

```
'mail:send {user} {--queue=default}'
```

Option Shortcuts

To assign a shortcut when defining an option, you may specify it before the option name and use the `|` character as a delimiter to separate the shortcut from the full option name:

```
'mail:send {user} {--Q|queue}'
```

When invoking the command on your terminal, option shortcuts should be prefixed with a single hyphen and no `=` character should be included when specifying a value for the option:

```
php artisan mail:send 1 -Qdefault
```

Input Arrays

If you would like to define arguments or options to expect multiple input values, you may use the `*` character. First, let's take a look at an example that specifies such an argument:

```
'mail:send {user*}'
```

When calling this method, the `user` arguments may be passed in order to the command line. For example, the following command will set the value of `user` to an array with `1` and `2` as its values:

```
php artisan mail:send 1 2
```

This `*` character can be combined with an optional argument definition to allow zero or more instances of an argument:

```
'mail:send {user?*}'
```

Option Arrays

When defining an option that expects multiple input values, each option value passed to the command should be prefixed with the option name:

```
'mail:send {--id=*}'
```

Such a command may be invoked by passing multiple `--id` arguments:

```
php artisan mail:send --id=1 --id=2
```

Input Descriptions

You may assign descriptions to input arguments and options by separating the argument name from the description using a colon. If you need a little extra room to define your command, feel free to spread the definition across multiple lines:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send
    {user : The ID of the user}
    {--queue : Whether the job should be queued}';
```

Prompting for Missing Input

If your command contains required arguments, the user will receive an error message when they are not provided.

Alternatively, you may configure your command to automatically prompt the user when required arguments are missing by implementing the `PromptsForMissingInput` interface:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\PromptsForMissingInput;

class SendEmails extends Command implements PromptsForMissingInput
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    // ...
}
```

If Laravel needs to gather a required argument from the user, it will automatically ask the user for the argument by intelligently phrasing the question using either the argument name or description. If you wish to customize the question used to gather the required argument, you may implement the `promptForMissingArgumentsUsing` method, returning an array of questions keyed by the argument names:

```
/**
 * Prompt for missing input arguments using the returned questions.
 *
 * @return array
 */
protected function promptForMissingArgumentsUsing()
{
    return [
        'user' => 'Which user ID should receive the mail?',
    ];
}
```

You may also provide placeholder text by using a tuple containing the question and placeholder:

```
return [
    'user' => ['Which user ID should receive the mail?', 'E.g. 123'],
];
```

If you would like complete control over the prompt, you may provide a closure that should prompt the user and return their answer:

```
use App\Models\User;
use function Laravel\Prompts\search;

// ...

return [
    'user' => fn () => search(
        label: 'Search for a user:',
        placeholder: 'E.g. Taylor Otwell',
        options: fn ($value) => strlen($value) > 0
            ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
            : []
    ),
];

```



The comprehensive [Laravel Prompts](#) documentation includes additional information on the available prompts and their usage.

If you wish to prompt the user to select or enter `options`, you may include prompts in your command's `handle` method. However, if you only wish to prompt the user when they have also been automatically prompted for missing arguments, then you may implement the `afterPromptingForMissingArguments` method:

```
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use function Laravel\Prompts\confirm;

// ...

/**
 * Perform actions after the user was prompted for missing arguments.
 *
 * @param \Symfony\Component\Console\Input\InputInterface $input
 * @param \Symfony\Component\Console\Output\OutputInterface $output
 * @return void
 */
protected function afterPromptingForMissingArguments(InputInterface $input, OutputInterface $output)
{
    $input->setOption('queue', confirm(
        label: 'Would you like to queue the mail?',
        default: $this->option('queue')
    ));
}
```

Command I/O

Retrieving Input

While your command is executing, you will likely need to access the values for the arguments and options accepted by your command. To do so, you may use the `argument` and `option` methods. If an argument or option does not exist, `null` will be returned:

```
/**
 * Execute the console command.
 */
public function handle(): void
{
    $userId = $this->argument('user');
}
```

If you need to retrieve all of the arguments as an `array`, call the `arguments` method:

```
$arguments = $this->arguments();
```

Options may be retrieved just as easily as arguments using the `option` method. To retrieve all of the options as an array, call the `options` method:

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options as an array...
$options = $this->options();
```

Prompting for Input



Laravel Prompts is a PHP package for adding beautiful and user-friendly forms to your command-line applications, with browser-like features including placeholder text and validation.

In addition to displaying output, you may also ask the user to provide input during the execution of your command. The `ask` method will prompt the user with the given question, accept their input, and then return the user's input back to your command:

```
/*
 * Execute the console command.
 */
public function handle(): void
{
    $name = $this->ask('What is your name?');

    // ...
}
```

The `ask` method also accepts an optional second argument which specifies the default value that should be returned if no user input is provided:

```
$name = $this->ask('What is your name?', 'Taylor');
```

The `secret` method is similar to `ask`, but the user's input will not be visible to them as they type in the console. This method is useful when asking for sensitive information such as passwords:

```
$password = $this->secret('What is the password?');
```

Asking for Confirmation

If you need to ask the user for a simple "yes or no" confirmation, you may use the `confirm` method. By default, this method will return `false`. However, if the user enters `y` or `yes` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    // ...
}
```

If necessary, you may specify that the confirmation prompt should return `true` by default by passing `true` as the second argument to the `confirm` method:

```
if ($this->confirm('Do you wish to continue?', true)) {
    // ...
}
```

Auto-Completion

The `anticipate` method can be used to provide auto-completion for possible choices. The user can still provide any answer, regardless of the auto-completion hints:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Alternatively, you may pass a closure as the second argument to the `anticipate` method. The closure will be called each time the user types an input character. The closure should accept a string parameter containing the user's input so far, and return an array of options for auto-completion:

```
$name = $this->anticipate('What is your address?', function (string $input) {
    // Return auto-completion options...
});
```

Multiple Choice Questions

If you need to give the user a predefined set of choices when asking a question, you may use the `choice` method. You may set the array index of the default value to be returned if no option is chosen by passing the index as the third argument to the method:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex
);
```

In addition, the `choice` method accepts optional fourth and fifth arguments for determining the maximum number of attempts to select a valid response and whether multiple selections are permitted:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex,
    $maxAttempts = null,
    $allowMultipleSelections = false
);
```

Writing Output

To send output to the console, you may use the `line`, `info`, `comment`, `question`, `warn`, and `error` methods. Each of these methods will use appropriate ANSI colors for their purpose. For example, let's display some general information to the user. Typically, the `info` method will display in the console as green colored text:

```
/** 
 * Execute the console command.
 */
public function handle(): void
{
    // ...

    $this->info('The command was successful!');
}
```

To display an error message, use the `error` method. Error message text is typically displayed in red:

```
$this->error('Something went wrong!');
```

You may use the `line` method to display plain, uncolored text:

```
$this->line('Display this on the screen');
```

You may use the `newLine` method to display a blank line:

```
// Write a single blank line...
$this->newLine();

// Write three blank lines...
$this->newLine(3);
```

Tables

The `table` method makes it easy to correctly format multiple rows / columns of data. All you need to do is provide the column names and the data for the table and Laravel will automatically calculate the appropriate width and height of the table for you:

```
use App\Models\User;

$this->table(
    ['Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

Progress Bars

For long running tasks, it can be helpful to show a progress bar that informs users how complete the task is. Using the `withProgressBar` method, Laravel will display a progress bar and advance its progress for each iteration over a given iterable value:

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function (User $user) {
    $this->performTask($user);
});
```

Sometimes, you may need more manual control over how a progress bar is advanced. First, define the total number of steps the process will iterate through. Then, advance the progress bar after processing each item:

```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```



For more advanced options, check out the [Symfony Progress Bar component documentation](#).

Registering Commands

All of your console commands are registered within your application's `App\Console\Kernel` class, which is your application's "console kernel". Within the `commands` method of this class, you will see a call to the kernel's `load` method. The `load` method will scan the `app/Console/Commands` directory and automatically register each command it contains with Artisan. You are even free to make additional calls to the `load` method to scan other directories for Artisan commands:

```
/**
 * Register the commands for the application.
 */

protected function commands(): void
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/../Domain/Orders/Commands');

    // ...
}
```

If necessary, you may manually register commands by adding the command's class name to a `$commands` property within your `App\Console\Kernel` class. If this property is not already defined on your kernel, you should define it manually. When Artisan boots, all the commands listed in this property will be resolved by the [service container](#) and registered with Artisan:

```
protected $commands = [
    Commands\SendEmails::class
];
```

Programmatically Executing Commands

Sometimes you may wish to execute an Artisan command outside of the CLI. For example, you may wish to execute an Artisan command from a route or controller. You may use the `call` method on the `Artisan` facade to accomplish this. The `call` method accepts either the command's signature name or class name as its first argument, and an array of command parameters as the second argument. The exit code will be returned:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

Alternatively, you may pass the entire Artisan command to the `call` method as a string:

```
Artisan::call('mail:send 1 --queue=default');
```

Passing Array Values

If your command defines an option that accepts an array, you may pass an array of values to that option:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

Passing Boolean Values

If you need to specify the value of an option that does not accept string values, such as the `--force` flag on the `migrate:refresh` command, you should pass `true` or `false` as the value of the option:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Queueing Artisan Commands

Using the `queue` method on the `Artisan` facade, you may even queue Artisan commands so they are processed in the background by your [queue workers](#). Before using this method, make sure you have configured your queue and are running a queue listener:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

Using the `onConnection` and `onQueue` methods, you may specify the connection or queue the Artisan command should be dispatched to:

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Calling Commands From Other Commands

Sometimes you may wish to call other commands from an existing Artisan command. You may do so using the `call` method. This `call` method accepts the command name and an array of command arguments / options:

```
/**
 * Execute the console command.
 */
public function handle(): void
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    // ...
}
```

If you would like to call another console command and suppress all of its output, you may use the `callSilently` method. The `callSilently` method has the same signature as the `call` method:

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Signal Handling

As you may know, operating systems allow signals to be sent to running processes. For example, the `SIGTERM` signal is how operating systems ask a program to terminate. If you wish to listen for signals in your Artisan console commands and execute code when they occur, you may use the `trap` method:

```
/**
 * Execute the console command.
 */
public function handle(): void
{
    $this->trap(SIGTERM, fn () => $this->shouldKeepRunning = false);

    while ($this->shouldKeepRunning) {
        // ...
    }
}
```

To listen for multiple signals at once, you may provide an array of signals to the `trap` method:

```
$this->trap([SIGTERM, SIGQUIT], function (int $signal) {
    $this->shouldKeepRunning = false;

    dump($signal); // SIGTERM / SIGQUIT
});
```

Stub Customization

The Artisan console's `make` commands are used to create a variety of classes, such as controllers, jobs, migrations, and tests. These classes are generated using "stub" files that are populated with values based on your input. However, you may want to make small changes to files generated by Artisan. To accomplish this, you may use the `stub:publish` command to publish the most common stubs to your application so that you can customize them:

```
php artisan stub:publish
```

The published stubs will be located within a `stubs` directory in the root of your application. Any changes you make to these stubs will be reflected when you generate their corresponding classes using Artisan's `make` commands.

Events

Artisan dispatches three events when running commands: `Illuminate\Console\Events\ArtisanStarting`, `Illuminate\Console\Events\CommandStarting`, and `Illuminate\Console\Events\CommandFinished`. The `ArtisanStarting` event is dispatched immediately when Artisan starts running. Next, the `CommandStarting` event is dispatched immediately before a command runs. Finally, the `CommandFinished` event is dispatched once a command finishes executing.

Broadcasting

- [Introduction](#)
- [Server Side Installation](#)
 - [Configuration](#)
 - [Reverb](#)
 - [Pusher Channels](#)
 - [Ably](#)
 - [Open Source Alternatives](#)
- [Client Side Installation](#)
 - [Reverb](#)
 - [Pusher Channels](#)
 - [Ably](#)
- [Concept Overview](#)
 - [Using an Example Application](#)
- [Defining Broadcast Events](#)
 - [Broadcast Name](#)
 - [Broadcast Data](#)
 - [Broadcast Queue](#)
 - [Broadcast Conditions](#)
 - [Broadcasting and Database Transactions](#)
- [Authorizing Channels](#)
 - [Defining Authorization Routes](#)
 - [Defining Authorization Callbacks](#)
 - [Defining Channel Classes](#)
- [Broadcasting Events](#)
 - [Only to Others](#)
 - [Customizing the Connection](#)
- [Receiving Broadcasts](#)
 - [Listening for Events](#)
 - [Leaving a Channel](#)
 - [Namespaces](#)
- [Presence Channels](#)
 - [Authorizing Presence Channels](#)
 - [Joining Presence Channels](#)
 - [Broadcasting to Presence Channels](#)
- [Model Broadcasting](#)
 - [Model Broadcasting Conventions](#)
 - [Listening for Model Broadcasts](#)
- [Client Events](#)
- [Notifications](#)

Introduction

In many modern web applications, WebSockets are used to implement realtime, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a WebSocket connection to be handled by the client. WebSockets provide a more efficient alternative to continually polling your application's server for data changes that should be reflected in your UI.

For example, imagine your application is able to export a user's data to a CSV file and email it to them. However, creating this CSV file takes several minutes so you choose to create and mail the CSV within a [queued job](#). When the CSV has been

created and mailed to the user, we can use event broadcasting to dispatch an `App\Events\UserDataExported` event that is received by our application's JavaScript. Once the event is received, we can display a message to the user that their CSV has been emailed to them without them ever needing to refresh the page.

To assist you in building these types of features, Laravel makes it easy to "broadcast" your server-side Laravel [events](#) over a WebSocket connection. Broadcasting your Laravel events allows you to share the same event names and data between your server-side Laravel application and your client-side JavaScript application.

The core concepts behind broadcasting are simple: clients connect to named channels on the frontend, while your Laravel application broadcasts events to these channels on the backend. These events can contain any additional data you wish to make available to the frontend.

Supported Drivers

By default, Laravel includes three server-side broadcasting drivers for you to choose from: [Laravel Reverb](#), [Pusher Channels](#), and [Ably](#).



Before diving into event broadcasting, make sure you have read Laravel's documentation on [events](#) and [listeners](#).

Server Side Installation

To get started using Laravel's event broadcasting, we need to do some configuration within the Laravel application as well as install a few packages.

Event broadcasting is accomplished by a server-side broadcasting driver that broadcasts your Laravel events so that Laravel Echo (a JavaScript library) can receive them within the browser client. Don't worry - we'll walk through each part of the installation process step-by-step.

Configuration

All of your application's event broadcasting configuration is stored in the `config/broadcasting.php` configuration file. Laravel supports several broadcast drivers out of the box: [Pusher Channels](#), [Redis](#), and a `log` driver for local development and debugging. Additionally, a `null` driver is included which allows you to totally disable broadcasting during testing. A configuration example is included for each of these drivers in the `config/broadcasting.php` configuration file.

Broadcast Service Provider

Before broadcasting any events, you will first need to register the `App\Providers\BroadcastServiceProvider`. In new Laravel applications, you only need to uncomment this provider in the `providers` array of your `config/app.php` configuration file. This `BroadcastServiceProvider` contains the code necessary to register the broadcast authorization routes and callbacks.

Queue Configuration

You will also need to configure and run a [queue worker](#). All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected by events being broadcast.

Reverb

You may install Reverb using the Composer package manager:

```
composer require laravel/reverb
```

Once the package is installed, you may run Reverb's installation command to publish the configuration, update your application's broadcasting configuration, and add Reverb's required environment variables:

```
php artisan reverb:install
```

You can find detailed Reverb installation and usage instructions in the [Reverb documentation](#).

Pusher Channels

If you plan to broadcast your events using [Pusher Channels](#), you should install the Pusher Channels PHP SDK using the Composer package manager:

```
composer require pusher/pusher-php-server
```

Next, you should configure your Pusher Channels credentials in the `config/broadcasting.php` configuration file. An example Pusher Channels configuration is already included in this file, allowing you to quickly specify your key, secret, and application ID. Typically, these values should be set via the `PUSHER_APP_KEY`, `PUSHER_APP_SECRET`, and `PUSHER_APP_ID` [environment variables](#):

```
PUSHER_APP_ID=your-pusher-app-id
PUSHER_APP_KEY=your-pusher-key
PUSHER_APP_SECRET=your-pusher-secret
PUSHER_APP_CLUSTER=mt1
```

The `config/broadcasting.php` file's `pusher` configuration also allows you to specify additional [options](#) that are supported by Channels, such as the cluster.

Next, you will need to change your broadcast driver to `pusher` in your `.env` file:

```
BROADCAST_DRIVER=pusher
```

Finally, you are ready to install and configure [Laravel Echo](#), which will receive the broadcast events on the client-side.

Open Source Pusher Alternatives

[soketi](#) provides a Pusher compatible WebSocket server for Laravel, allowing you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using open source packages for broadcasting, please consult our documentation on [open source alternatives](#).

Ably



The documentation below discusses how to use Ably in "Pusher compatibility" mode. However, the Ably team recommends and maintains a broadcaster and Echo client that is able to take advantage of the unique capabilities offered by Ably. For more information on using the Ably maintained drivers, please [consult Ably's Laravel broadcaster documentation](#).

If you plan to broadcast your events using [Ably](#), you should install the Ably PHP SDK using the Composer package manager:

```
composer require ably/ably-php
```

Next, you should configure your Ably credentials in the `config/broadcasting.php` configuration file. An example Ably configuration is already included in this file, allowing you to quickly specify your key. Typically, this value should be set via the `ABLY_KEY` environment variable:

```
ABLY_KEY=your-ably-key
```

Next, you will need to change your broadcast driver to `ably` in your `.env` file:

```
BROADCAST_DRIVER=ably
```

Finally, you are ready to install and configure [Laravel Echo](#), which will receive the broadcast events on the client-side.

Open Source Alternatives

Node

[Soketi](#) is a Node based, Pusher compatible WebSocket server for Laravel. Under the hood, Soketi utilizes µWebSockets.js for extreme scalability and speed. This package allows you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using this package, please consult its [official documentation](#).

Client Side Installation

Reverb

[Laravel Echo](#) is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package since Reverb utilizes the Pusher protocol for WebSocket subscriptions, channels, and messages:

```
npm install --save-dev laravel-echo pusher-js
```

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file - you simply need to uncomment it and update the `broadcaster` configuration option to `reverb`:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'reverb',
    key: import.meta.env.VITE_REVERB_APP_KEY,
    wsHost: import.meta.env.VITE_REVERB_HOST,
    wsPort: import.meta.env.VITE_REVERB_PORT,
    wssPort: import.meta.env.VITE_REVERB_PORT,
    forceTLS: (import.meta.env.VITE_REVERB_SCHEME ?? 'https') === 'https',
    enabledTransports: ['ws', 'wss'],
});
```

Next, you should compile your application's assets:

```
npm run build
```



The Laravel Echo `reverb` broadcaster requires `laravel-echo v1.16.0+`.

Pusher Channels

[Laravel Echo](#) is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package since we will be using the Pusher Channels broadcaster:

```
npm install --save-dev laravel-echo pusher-js
```

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file - you simply need to uncomment it:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run build
```



To learn more about compiling your application's JavaScript assets, please consult the documentation on [Vite](#).

Using an Existing Client Instance

If you already have a pre-configured Pusher Channels client instance that you would like Echo to utilize, you may pass it to Echo via the `client` configuration option:

```

import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key'
}

window.Echo = new Echo({
    ...options,
    client: new Pusher(options.key, options)
});

```

Ably



The documentation below discusses how to use Ably in "Pusher compatibility" mode. However, the Ably team recommends and maintains a broadcaster and Echo client that is able to take advantage of the unique capabilities offered by Ably. For more information on using the Ably maintained drivers, please [consult Ably's Laravel broadcaster documentation](#).

[Laravel Echo](#) is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package.

You may wonder why we would install the `pusher-js` JavaScript library even though we are using Ably to broadcast our events. Thankfully, Ably includes a Pusher compatibility mode which lets us use the Pusher protocol when listening for events in our client-side application:

```
npm install --save-dev laravel-echo pusher-js
```

Before continuing, you should enable Pusher protocol support in your Ably application settings. You may enable this feature within the "Protocol Adapter Settings" portion of your Ably application's settings dashboard.

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file; however, the default configuration in the `bootstrap.js` file is intended for Pusher. You may copy the configuration below to transition your configuration to Ably:

```

import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_ABLY_PUBLIC_KEY,
    wsHost: 'realtime-pusher.ably.io',
    wsPort: 443,
    disableStats: true,
    encrypted: true,
});

```

Note that our Ably Echo configuration references a `VITE_ABLY_PUBLIC_KEY` environment variable. This variable's value should be your Ably public key. Your public key is the portion of your Ably key that occurs before the `:` character.

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run dev
```



To learn more about compiling your application's JavaScript assets, please consult the documentation on [Vite](#).

Concept Overview

Laravel's event broadcasting allows you to broadcast your server-side Laravel events to your client-side JavaScript application using a driver-based approach to WebSockets. Currently, Laravel ships with [Pusher Channels](#) and [Ably](#) drivers. The events may be easily consumed on the client-side using the [Laravel Echo](#) JavaScript package.

Events are broadcast over "channels", which may be specified as public or private. Any visitor to your application may subscribe to a public channel without any authentication or authorization; however, in order to subscribe to a private channel, a user must be authenticated and authorized to listen on that channel.



If you would like to explore open source alternatives to Pusher, check out the [open source alternatives](#).

Using an Example Application

Before diving into each component of event broadcasting, let's take a high level overview using an e-commerce store as an example.

In our application, let's assume we have a page that allows users to view the shipping status for their orders. Let's also assume that an `OrderShipmentStatusUpdated` event is fired when a shipping status update is processed by the application:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

The `ShouldBroadcast` Interface

When a user is viewing one of their orders, we don't want them to have to refresh the page to view status updates. Instead, we want to broadcast the updates to the application as they are created. So, we need to mark the `OrderShipmentStatusUpdated` event with the `ShouldBroadcast` interface. This will instruct Laravel to broadcast the event when it is fired:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;
}
```

The `ShouldBroadcast` interface requires our event to define a `broadcastOn` method. This method is responsible for returning the channels that the event should broadcast on. An empty stub of this method is already defined on generated event classes, so we only need to fill in its details. We only want the creator of the order to be able to view status updates, so we will broadcast the event on a private channel that is tied to the order:

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channel the event should broadcast on.
 */
public function broadcastOn(): Channel
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

If you wish the event to broadcast on multiple channels, you may return an `array` instead:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PrivateChannel('orders.'.$this->order->id),
        // ...
    ];
}
```

Authorizing Channels

Remember, users must be authorized to listen on private channels. We may define our channel authorization rules in our application's `routes/channels.php` file. In this example, we need to verify that any user attempting to listen on the private `orders.1` channel is actually the creator of the order:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Listening for Event Broadcasts

Next, all that remains is to listen for the event in our JavaScript application. We can do this using [Laravel Echo](#). First, we'll use the `private` method to subscribe to the private channel. Then, we may use the `listen` method to listen for the `OrderShipmentStatusUpdated` event. By default, all of the event's public properties will be included on the broadcast event:

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
   });
```

Defining Broadcast Events

To inform Laravel that a given event should be broadcast, you must implement the `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface on the event class. This interface is already imported into all event classes generated by the framework so you may easily add it to any of your events.

The `ShouldBroadcast` interface requires you to implement a single method: `broadcastOn`. The `broadcastOn` method should return a channel or array of channels that the event should broadcast on. The channels should be instances of `Channel`, `PrivateChannel`, or `PresenceChannel`. Instances of `Channel` represent public channels that any user may subscribe to, while `PrivateChannels` and `PresenceChannels` represent private channels that require [channel authorization](#):

```
<?php

namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * Create a new event instance.
     */
    public function __construct(
        public User $user,
    ) {}

    /**
     * Get the channels the event should broadcast on.
     *
     * @return array<int, \Illuminate\Broadcasting\Channel>
     */
    public function broadcastOn(): array
    {
        return [
            new PrivateChannel('user.'.$this->user->id),
        ];
    }
}
```

After implementing the `ShouldBroadcast` interface, you only need to [fire the event](#) as you normally would. Once the event has been fired, a [queued job](#) will automatically broadcast the event using your specified broadcast driver.

Broadcast Name

By default, Laravel will broadcast the event using the event's class name. However, you may customize the broadcast name by defining a `broadcastAs` method on the event:

```
/**
 * The event's broadcast name.
 */
public function broadcastAs(): string
{
    return 'server.created';
}
```

If you customize the broadcast name using the `broadcastAs` method, you should make sure to register your listener with a leading `.` character. This will instruct Echo to not prepend the application's namespace to the event:

```
.listen('.server.created', function (e) {
    ...
});
```

Broadcast Data

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

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

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

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

Broadcast Queue

By default, each broadcast event is placed on the default queue for the default queue connection specified in your `queue.php` configuration file. You may customize the queue connection and name used by the broadcaster by defining `connection` and `queue` properties on your event class:

```
/**
 * The name of the queue connection to use when broadcasting the event.
 *
 * @var string
 */
public $connection = 'redis';

/**
 * The name of the queue on which to place the broadcasting job.
 *
 * @var string
 */
public $queue = 'default';
```

Alternatively, you may customize the queue name by defining a `broadcastQueue` method on your event:

```
/**
 * The name of the queue on which to place the broadcasting job.
 */
public function broadcastQueue(): string
{
    return 'default';
}
```

If you would like to broadcast your event using the `sync` queue instead of the default queue driver, you can implement the `ShouldBroadcastNow` interface instead of `ShouldBroadcast`:

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class OrderShipmentStatusUpdated implements ShouldBroadcastNow
{
    // ...
}
```

Broadcast Conditions

Sometimes you want to broadcast your event only if a given condition is true. You may define these conditions by adding a `broadcastWhen` method to your event class:

```
/**
 * Determine if this event should broadcast.
 */
public function broadcastWhen(): bool
{
    return $this->order->value > 100;
}
```

Broadcasting and Database Transactions

When broadcast events are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your event depends on these models, unexpected errors can occur when the job that broadcasts the event is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular broadcast event should be dispatched after all open database transactions have been committed by implementing the `ShouldDispatchAfterCommit` interface on the event class:

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Contracts\Events\ShouldDispatchAfterCommit;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast, ShouldDispatchAfterCommit
{
    use SerializesModels;
}
```



To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).

Authorizing Channels

Private channels require you to authorize that the currently authenticated user can actually listen on the channel. This is accomplished by making an HTTP request to your Laravel application with the channel name and allowing your application to determine if the user can listen on that channel. When using [Laravel Echo](#), the HTTP request to authorize subscriptions to private channels will be made automatically; however, you do need to define the proper routes to respond to these requests.

Defining Authorization Routes

Thankfully, Laravel makes it easy to define the routes to respond to channel authorization requests. In the `App\Providers\BroadcastServiceProvider` included with your Laravel application, you will see a call to the `Broadcast::routes` method. This method will register the `/broadcasting/auth` route to handle authorization requests:

```
Broadcast::routes();
```

The `Broadcast::routes` method will automatically place its routes within the `web` middleware group; however, you may pass an array of route attributes to the method if you would like to customize the assigned attributes:

```
Broadcast::routes($attributes);
```

Customizing the Authorization Endpoint

By default, Echo will use the `/broadcasting/auth` endpoint to authorize channel access. However, you may specify your own authorization endpoint by passing the `authEndpoint` configuration option to your Echo instance:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    authEndpoint: '/custom/endpoint/auth'
});
```

Customizing the Authorization Request

You can customize how Laravel Echo performs authorization requests by providing a custom authorizer when initializing Echo:

```
window.Echo = new Echo({
    // ...
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                    .then(response => {
                        callback(null, response.data);
                    })
                    .catch(error => {
                        callback(error);
                    });
            };
        };
    },
});
```

Defining Authorization Callbacks

Next, we need to define the logic that will actually determine if the currently authenticated user can listen to a given channel. This is done in the `routes/channels.php` file that is included with your application. In this file, you may use the `Broadcast::channel` method to register channel authorization callbacks:

```
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

You may view a list of your application's broadcast authorization callbacks using the `channel:list` Artisan command:

```
php artisan channel:list
```

Authorization Callback Model Binding

Just like HTTP routes, channel routes may also take advantage of implicit and explicit [route model binding](#). For example, instead of receiving a string or numeric order ID, you may request an actual `Order` model instance:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{order}', function (User $user, Order $order) {
    return $user->id === $order->user_id;
});
```



Unlike HTTP route model binding, channel model binding does not support automatic [implicit model binding scoping](#). However, this is rarely a problem because most channels can be scoped based on a single model's unique, primary key.

Authorization Callback Authentication

Private and presence broadcast channels authenticate the current user via your application's default authentication guard. If the user is not authenticated, channel authorization is automatically denied and the authorization callback is never executed. However, you may assign multiple, custom guards that should authenticate the incoming request if necessary:

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]);
```

Defining Channel Classes

If your application is consuming many different channels, your `routes/channels.php` file could become bulky. So, instead of using closures to authorize channels, you may use channel classes. To generate a channel class, use the `make:channel` Artisan command. This command will place a new channel class in the `App/Broadcasting` directory.

```
php artisan make:channel OrderChannel
```

Next, register your channel in your `routes/channels.php` file:

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

Finally, you may place the authorization logic for your channel in the channel class' `join` method. This `join` method will house the same logic you would have typically placed in your channel authorization closure. You may also take advantage of channel model binding:

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * Create a new channel instance.
     */
    public function __construct()
    {
        // ...
    }

    /**
     * Authenticate the user's access to the channel.
     */
    public function join(User $user, Order $order): array|bool
    {
        return $user->id === $order->user_id;
    }
}
```



Like many other classes in Laravel, channel classes will automatically be resolved by the [service container](#). So, you may type-hint any dependencies required by your channel in its constructor.

Broadcasting Events

Once you have defined an event and marked it with the `ShouldBroadcast` interface, you only need to fire the event using the event's `dispatch` method. The event dispatcher will notice that the event is marked with the `ShouldBroadcast` interface and will queue the event for broadcasting:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

Only to Others

When building an application that utilizes event broadcasting, you may occasionally need to broadcast an event to all subscribers to a given channel except for the current user. You may accomplish this using the `broadcast` helper and the `toOthers` method:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

To better understand when you may want to use the `toOthers` method, let's imagine a task list application where a user may create a new task by entering a task name. To create a task, your application might make a request to a `/task` URL which broadcasts the task's creation and returns a JSON representation of the new task. When your JavaScript application receives the response from the end-point, it might directly insert the new task into its task list like so:

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
 });
```

However, remember that we also broadcast the task's creation. If your JavaScript application is also listening for this event in order to add tasks to the task list, you will have duplicate tasks in your list: one from the end-point and one from the broadcast. You may solve this by using the `toOthers` method to instruct the broadcaster to not broadcast the event to the current user.



Your event must use the `Illuminate\Broadcasting\InteractsWithSockets` trait in order to call the `toOthers` method.

Configuration

When you initialize a Laravel Echo instance, a socket ID is assigned to the connection. If you are using a global `Axios` instance to make HTTP requests from your JavaScript application, the socket ID will automatically be attached to every outgoing request as an `X-Socket-ID` header. Then, when you call the `toOthers` method, Laravel will extract the socket ID from the header and instruct the broadcaster to not broadcast to any connections with that socket ID.

If you are not using a global Axios instance, you will need to manually configure your JavaScript application to send the `X-Socket-ID` header with all outgoing requests. You may retrieve the socket ID using the `Echo.socketId` method:

```
var socketId = Echo.socketId();
```

Customizing the Connection

If your application interacts with multiple broadcast connections and you want to broadcast an event using a broadcaster other than your default, you may specify which connection to push an event to using the `via` method:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->via('pusher');
```

Alternatively, you may specify the event's broadcast connection by calling the `broadcastVia` method within the event's constructor. However, before doing so, you should ensure that the event class uses the `InteractsWithBroadcasting` trait:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithBroadcasting;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    use InteractsWithBroadcasting;

    /**
     * Create a new event instance.
     */
    public function __construct()
    {
        $this->broadcastVia('pusher');
    }
}
```

Receiving Broadcasts

Listening for Events

Once you have [installed and instantiated Laravel Echo](#), you are ready to start listening for events that are broadcast from your Laravel application. First, use the `channel` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event:

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
});
```

If you would like to listen for events on a private channel, use the `private` method instead. You may continue to chain calls to the `listen` method to listen for multiple events on a single channel:

```
Echo.private(`orders.${this.order.id}`)
    .listen(/* ... */)
    .listen(/* ... */)
    .listen(/* ... */);
```

Stop Listening for Events

If you would like to stop listening to a given event without [leaving the channel](#), you may use the `stopListening` method:

```
Echo.private(`orders.${this.order.id}`)
    .stopListening('OrderShipmentStatusUpdated')
```

Leaving a Channel

To leave a channel, you may call the `leaveChannel` method on your Echo instance:

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

If you would like to leave a channel and also its associated private and presence channels, you may call the `leave` method:

```
Echo.leave(`orders.${this.order.id}`);
```

Namespaces

You may have noticed in the examples above that we did not specify the full `App\Events` namespace for the event classes. This is because Echo will automatically assume the events are located in the `App\Events` namespace. However, you may configure the root namespace when you instantiate Echo by passing a `namespace` configuration option:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    namespace: 'App.Other.Namespace'
});
```

Alternatively, you may prefix event classes with a `.` when subscribing to them using Echo. This will allow you to always specify the fully-qualified class name:

```
Echo.channel('orders')
    .listen('.Namespace\\Event\\Class', (e) => {
        // ...
});
```

Presence Channels

Presence channels build on the security of private channels while exposing the additional feature of awareness of who is subscribed to the channel. This makes it easy to build powerful, collaborative application features such as notifying users when another user is viewing the same page or listing the inhabitants of a chat room.

Authorizing Presence Channels

All presence channels are also private channels; therefore, users must be [authorized to access them](#). However, when defining authorization callbacks for presence channels, you will not return `true` if the user is authorized to join the channel. Instead, you should return an array of data about the user.

The data returned by the authorization callback will be made available to the presence channel event listeners in your JavaScript application. If the user is not authorized to join the presence channel, you should return `false` or `null`:

```
use App\Models\User;

Broadcast::channel('chat.{roomId}', function (User $user, int $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

Joining Presence Channels

To join a presence channel, you may use Echo's `join` method. The `join` method will return a `PresenceChannel` implementation which, along with exposing the `listen` method, allows you to subscribe to the `here`, `joining`, and `leaving` events.

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        // ...
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    })
    .error((error) => {
        console.error(error);
    });
});
```

The `here` callback will be executed immediately once the channel is joined successfully, and will receive an array containing the user information for all of the other users currently subscribed to the channel. The `joining` method will be executed when a new user joins a channel, while the `leaving` method will be executed when a user leaves the channel. The `error` method will be executed when the authentication endpoint returns an HTTP status code other than 200 or if there is a problem parsing the returned JSON.

Broadcasting to Presence Channels

Presence channels may receive events just like public or private channels. Using the example of a chatroom, we may want to broadcast `NewMessage` events to the room's presence channel. To do so, we'll return an instance of `PresenceChannel` from the event's `broadcastOn` method:

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PresenceChannel('chat.' . $this->message->room_id),
    ];
}
```

As with other events, you may use the `broadcast` helper and the `toOthers` method to exclude the current user from receiving the broadcast:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

As typical of other types of events, you may listen for events sent to presence channels using Echo's `listen` method:

```
Echo.join(`chat.${roomId}`)
    .here(/* ... */)
    .joining(/* ... */)
    .leaving(/* ... */)
    .listen('NewMessage', (e) => {
        // ...
    });
});
```

Model Broadcasting



Before reading the following documentation about model broadcasting, we recommend you become familiar with the general concepts of Laravel's model broadcasting services as well as how to manually create and listen to broadcast events.

It is common to broadcast events when your application's [Eloquent models](#) are created, updated, or deleted. Of course, this can easily be accomplished by manually [defining custom events for Eloquent model state changes](#) and marking those events with the `ShouldBroadcast` interface.

However, if you are not using these events for any other purposes in your application, it can be cumbersome to create event classes for the sole purpose of broadcasting them. To remedy this, Laravel allows you to indicate that an Eloquent model should automatically broadcast its state changes.

To get started, your Eloquent model should use the `Illuminate\Database\Eloquent\BroadcastsEvents` trait. In addition, the model should define a `broadcastOn` method, which will return an array of channels that the model's events should broadcast on:

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Post extends Model
{
    use BroadcastsEvents, HasFactory;

    /**
     * Get the user that the post belongs to.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }

    /**
     * Get the channels that model events should broadcast on.
     *
     * @return array<int, \Illuminate\Broadcasting\Channel|\Illuminate\Database\Eloquent\Model>
     */
    public function broadcastOn(string $event): array
    {
        return [$this, $this->user];
    }
}
```

Once your model includes this trait and defines its broadcast channels, it will begin automatically broadcasting events when a model instance is created, updated, deleted, trashed, or restored.

In addition, you may have noticed that the `broadcastOn` method receives a string `$event` argument. This argument contains the type of event that has occurred on the model and will have a value of `created`, `updated`, `deleted`, `trashed`, or `restored`. By inspecting the value of this variable, you may determine which channels (if any) the model should broadcast to for a particular event:

```
/**
 * Get the channels that model events should broadcast on.
 *
 * @return array<string, array<int,
 \Illuminate\Broadcasting\Channel|\Illuminate\Database\Eloquent\Model>>
 */
public function broadcastOn(string $event): array
{
    return match ($event) {
        'deleted' => [],
        default => [$this, $this->user],
    };
}
```

Customizing Model Broadcasting Event Creation

Occasionally, you may wish to customize how Laravel creates the underlying model broadcasting event. You may accomplish this by defining a `newBroadcastableEvent` method on your Eloquent model. This method should return an `Illuminate\Database\Eloquent\BroadcastableModelEventOccurred` instance:

```
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred;

/**
 * Create a new broadcastable model event for the model.
 */
protected function newBroadcastableEvent(string $event): BroadcastableModelEventOccurred
{
    return (new BroadcastableModelEventOccurred(
        $this, $event
    ))->dontBroadcastToCurrentUser();
}
```

Model Broadcasting Conventions

Channel Conventions

As you may have noticed, the `broadcastOn` method in the model example above did not return `Channel` instances. Instead, Eloquent models were returned directly. If an Eloquent model instance is returned by your model's `broadcastOn` method (or is contained in an array returned by the method), Laravel will automatically instantiate a private channel instance for the model using the model's class name and primary key identifier as the channel name.

So, an `App\Models\User` model with an `id` of `1` would be converted into an `Illuminate\Broadcasting\PrivateChannel` instance with a name of `App.Models.User.1`. Of course, in addition to returning Eloquent model instances from your model's `broadcastOn` method, you may return complete `Channel` instances in order to have full control over the model's channel names:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channels that model events should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(string $event): array
{
    return [
        new PrivateChannel('user.' . $this->id)
    ];
}
```

If you plan to explicitly return a channel instance from your model's `broadcastOn` method, you may pass an Eloquent model instance to the channel's constructor. When doing so, Laravel will use the model channel conventions discussed above to convert the Eloquent model into a channel name string:

```
return [new Channel($this->user)];
```

If you need to determine the channel name of a model, you may call the `broadcastChannel` method on any model instance. For example, this method returns the string `App.Models.User.1` for an `App\Models\User` model with an `id` of `1`:

```
$user->broadcastChannel()
```

Event Conventions

Since model broadcast events are not associated with an "actual" event within your application's `App\\Events` directory, they are assigned a name and a payload based on conventions. Laravel's convention is to broadcast the event using the class name of the model (not including the namespace) and the name of the model event that triggered the broadcast.

So, for example, an update to the `App\Models\Post` model would broadcast an event to your client-side application as `PostUpdated` with the following payload:

```
{
    "model": {
        "id": 1,
        "title": "My first post"
        ...
    },
    ...
    "socket": "someSocketId",
}
```

The deletion of the `App\Models\User` model would broadcast an event named `UserDeleted`.

If you would like, you may define a custom broadcast name and payload by adding a `broadcastAs` and `broadcastWith` method to your model. These methods receive the name of the model event / operation that is occurring, allowing you to customize the event's name and payload for each model operation. If `null` is returned from the `broadcastAs` method, Laravel will use the model broadcasting event name conventions discussed above when broadcasting the event:

```
/**
 * The model event's broadcast name.
 */
public function broadcastAs(string $event): string|null
{
    return match ($event) {
        'created' => 'post.created',
        default => null,
    };
}

/**
 * Get the data to broadcast for the model.
 *
 * @return array<string, mixed>
 */
public function broadcastWith(string $event): array
{
    return match ($event) {
        'created' => ['title' => $this->title],
        default => ['model' => $this],
    };
}
```

Listening for Model Broadcasts

Once you have added the `BroadcastsEvents` trait to your model and defined your model's `broadcastOn` method, you are ready to start listening for broadcasted model events within your client-side application. Before getting started, you may wish to consult the complete documentation on [listening for events](#).

First, use the `private` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event. Typically, the channel name given to the `private` method should correspond to Laravel's [model broadcasting conventions](#).

Once you have obtained a channel instance, you may use the `listen` method to listen for a particular event. Since model broadcast events are not associated with an "actual" event within your application's `App\Events` directory, the `event name` must be prefixed with a `.` to indicate it does not belong to a particular namespace. Each model broadcast event has a `model` property which contains all of the broadcastable properties of the model:

```
Echo.private(`App.Models.User.${this.user.id}`)
    .listen('.PostUpdated', (e) => {
        console.log(e.model);
});
```

Client Events



When using [Pusher Channels](#), you must enable the "Client Events" option in the "App Settings" section of your [application dashboard](#) in order to send client events.

Sometimes you may wish to broadcast an event to other connected clients without hitting your Laravel application at all. This can be particularly useful for things like "typing" notifications, where you want to alert users of your application that another user is typing a message on a given screen.

To broadcast client events, you may use Echo's `whisper` method:

```
Echo.private(`chat.${roomId}`)
    .whisper('typing', {
        name: this.user.name
});
```

To listen for client events, you may use the `listenForWhisper` method:

```
Echo.private(`chat.${roomId}`)
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
});
```

Notifications

By pairing event broadcasting with [notifications](#), your JavaScript application may receive new notifications as they occur without needing to refresh the page. Before getting started, be sure to read over the documentation on [using the broadcast notification channel](#).

Once you have configured a notification to use the broadcast channel, you may listen for the broadcast events using Echo's `notification` method. Remember, the channel name should match the class name of the entity receiving the notifications:

```
Echo.private(`App.Models.User.${userId}`)
.notification((notification) => {
    console.log(notification.type);
});
```

In this example, all notifications sent to `App\Models\User` instances via the `broadcast` channel would be received by the callback. A channel authorization callback for the `App.Models.User.{id}` channel is included in the default `BroadcastServiceProvider` that ships with the Laravel framework.

Cache

- [Introduction](#)
- [Configuration](#)
 - [Driver Prerequisites](#)
- [Cache Usage](#)
 - [Obtaining a Cache Instance](#)
 - [Retrieving Items From the Cache](#)
 - [Storing Items in the Cache](#)
 - [Removing Items From the Cache](#)
 - [The Cache Helper](#)
- [Atomic Locks](#)
 - [Driver Prerequisites](#)
 - [Managing Locks](#)
 - [Managing Locks Across Processes](#)
- [Adding Custom Cache Drivers](#)
 - [Writing the Driver](#)
 - [Registering the Driver](#)
- [Events](#)

Introduction

Some of the data retrieval or processing tasks performed by your application could be CPU intensive or take several seconds to complete. When this is the case, it is common to cache the retrieved data for a time so it can be retrieved quickly on subsequent requests for the same data. The cached data is usually stored in a very fast data store such as [Memcached](#) or [Redis](#).

Thankfully, Laravel provides an expressive, unified API for various cache backends, allowing you to take advantage of their blazing fast data retrieval and speed up your web application.

Configuration

Your application's cache configuration file is located at `config/cache.php`. In this file, you may specify which cache driver you would like to be used by default throughout your application. Laravel supports popular caching backends like [Memcached](#), [Redis](#), [DynamoDB](#), and relational databases out of the box. In addition, a file based cache driver is available, while `array` and "null" cache drivers provide convenient cache backends for your automated tests.

The cache configuration file also contains various other options, which are documented within the file, so make sure to read over these options. By default, Laravel is configured to use the `file` cache driver, which stores the serialized, cached objects on the server's filesystem. For larger applications, it is recommended that you use a more robust driver such as Memcached or Redis. You may even configure multiple cache configurations for the same driver.

Driver Prerequisites

Database

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

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



You may also use the `php artisan cache:table` Artisan command to generate a migration with the proper schema.

Memcached

Using the Memcached driver requires the [Memcached PECL package](#) to be installed. You may list all of your Memcached servers in the `config/cache.php` configuration file. This file already contains a `memcached.servers` entry to get you started:

```
'memcached' => [
    'servers' => [
        [
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),
            'port' => env('MEMCACHED_PORT', 11211),
            'weight' => 100,
        ],
        [
        ],
    ],
],
```

If needed, you may set the `host` option to a UNIX socket path. If you do this, the `port` option should be set to `0`:

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

Before using a Redis cache with Laravel, you will need to either install the `PhpRedis` PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. [Laravel Sail](#) already includes this extension. In addition, official Laravel deployment platforms such as [Laravel Forge](#) and [Laravel Vapor](#) have the `PhpRedis` extension installed by default.

For more information on configuring Redis, consult its [Laravel documentation page](#).

DynamoDB

Before using the [DynamoDB](#) cache driver, you must create a DynamoDB table to store all of the cached data. Typically, this table should be named `cache`. However, you should name the table based on the value of the `stores.dynamodb.table` configuration value within your application's `cache` configuration file.

This table should also have a string partition key with a name that corresponds to the value of the `stores.dynamodb.attributes.key` configuration item within your application's `cache` configuration file. By default, the partition key should be named `key`.

Cache Usage

Obtaining a Cache Instance

To obtain a cache store instance, you may use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Laravel cache contracts:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

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

        return [
            // ...
        ];
    }
}
```

Accessing Multiple Cache Stores

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

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

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

Retrieving Items From the Cache

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

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

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

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

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

Determining Item Existence

The `has` method may be used to determine if an item exists in the cache. This method will also return `false` if the item exists but its value is `null`:

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

Incrementing / Decrementing Values

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

```
// Initialize the value if it does not exist...
Cache::add('key', 0, now()->addHours(4));

// Increment or decrement the value...
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

Retrieve and Store

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

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

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

You may use the `rememberForever` method to retrieve an item from the cache or store it forever if it does not exist:

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

Retrieve and Delete

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

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

Storing Items in the Cache

You may use the `put` method on the `Cache` facade to store items in the cache:

```
Cache::put('key', 'value', $seconds = 10);
```

If the storage time is not passed to the `put` method, the item will be stored indefinitely:

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

Instead of passing the number of seconds as an integer, you may also pass a `DateTime` instance representing the desired expiration time of the cached item:

```
Cache::put('key', 'value', now()>addMinutes(10));
```

Store if Not Present

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

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

Storing Items Forever

The `forever` method may be used to store an item in the cache permanently. Since these items will not expire, they must be manually removed from the cache using the `forget` method:

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



If you are using the Memcached driver, items that are stored "forever" may be removed when the cache reaches its size limit.

Removing Items From the Cache

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

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

You may also remove items by providing a zero or negative number of expiration seconds:

```
Cache::put('key', 'value', 0);
```

```
Cache::put('key', 'value', -5);
```

You may clear the entire cache using the `flush` method:

```
Cache::flush();
```



Flushing the cache does not respect your configured cache "prefix" and will remove all entries from the cache. Consider this carefully when clearing a cache which is shared by other applications.

The Cache Helper

In addition to using the `Cache` facade, you may also use the global `cache` function to retrieve and store data via the cache. When the `cache` function is called with a single, string argument, it will return the value of the given key:

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

If you provide an array of key / value pairs and an expiration time to the function, it will store values in the cache for the specified duration:

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

When the `cache` function is called without any arguments, it returns an instance of the `Illuminate\Contracts\Cache\Factory` implementation, allowing you to call other caching methods:

```
cache()->remember('users', $seconds, function () {  
    return DB::table('users')->get();  
});
```



When testing call to the global `cache` function, you may use the `Cache::shouldReceive` method just as if you were [testing the facade](#).

Atomic Locks



To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`, or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Driver Prerequisites

Database

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

```
Schema::create('cache_locks', function (Blueprint $table) {
    $table->string('key')->primary();
    $table->string('owner');
    $table->integer('expiration');
});
```



If you used the `cache:table` Artisan command to create the database driver's cache table, the migration created by that command already includes a definition for the `cache_locks` table.

Managing Locks

Atomic locks allow for the manipulation of distributed locks without worrying about race conditions. For example, [Laravel Forge](#) uses atomic locks to ensure that only one remote task is being executed on a server at a time. You may create and manage locks using the `Cache::lock` method:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...

    $lock->release();
}
```

The `get` method also accepts a closure. After the closure is executed, Laravel will automatically release the lock:

```
Cache::lock('foo', 10)->get(function () {
    // Lock acquired for 10 seconds and automatically released...
});
```

If the lock is not available at the moment you request it, you may instruct Laravel to wait for a specified number of seconds. If the lock can not be acquired within the specified time limit, an

`Illuminate\Contracts\Cache\LockTimeoutException` will be thrown:

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5);

    // Lock acquired after waiting a maximum of 5 seconds...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    $lock?->release();
}
```

The example above may be simplified by passing a closure to the `block` method. When a closure is passed to this method, Laravel will attempt to acquire the lock for the specified number of seconds and will automatically release the lock once the closure has been executed:

```
Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting a maximum of 5 seconds...
});
```

Managing Locks Across Processes

Sometimes, you may wish to acquire a lock in one process and release it in another process. For example, you may acquire a lock during a web request and wish to release the lock at the end of a queued job that is triggered by that request. In this scenario, you should pass the lock's scoped "owner token" to the queued job so that the job can re-instantiate the lock using the given token.

In the example below, we will dispatch a queued job if a lock is successfully acquired. In addition, we will pass the lock's owner token to the queued job via the lock's `owner` method:

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

Within our application's `ProcessPodcast` job, we can restore and release the lock using the owner token:

```
Cache::restoreLock('processing', $this->owner)->release();
```

If you would like to release a lock without respecting its current owner, you may use the `forceRelease` method:

```
Cache::lock('processing')->forceRelease();
```

Adding Custom Cache Drivers

Writing the Driver

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` [contract](#). So, a MongoDB cache implementation might look something like this:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

We just need to implement each of these methods using a MongoDB connection. For an example of how to implement each of these methods, take a look at the `Illuminate\Cache\MemcachedStore` in the [Laravel framework source code](#). Once our implementation is complete, we can finish our custom driver registration by calling the `Cache` facade's `extend` method:

```
Cache::extend('mongo', function (Application $app) {
    return Cache::repository(new MongoStore());
});
```



If you're wondering where to put your custom cache driver code, you could create an `Extensions` namespace within your `app` directory. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Registering the Driver

To register the custom cache driver with Laravel, we will use the `extend` method on the `Cache` facade. Since other service providers may attempt to read cached values within their `boot` method, we will register our custom driver within a `booting` callback. By using the `booting` callback, we can ensure that the custom driver is registered just before the `boot` method is called on our application's service providers but after the `register` method is called on all of the service providers. We will register our `booting` callback within the `register` method of our application's `App\Providers\AppServiceProvider` class:

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->booting(function () {
            Cache::extend('mongo', function (Application $app) {
                return Cache::repository(new MongoStore());
            });
        });
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        // ...
    }
}
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `config/cache.php` configuration file. The second argument is a closure that should return an `Illuminate\Cache\Repository` instance. The closure will be passed an `$app` instance, which is an instance of the [service container](#).

Once your extension is registered, update your `config/cache.php` configuration file's `driver` option to the name of your extension.

Events

To execute code on every cache operation, you may listen for the `events` fired by the cache. Typically, you should place these event listeners within your application's `App\Providers\EventServiceProvider` class:

```
use App\Listeners\LogCacheHit;
use App\Listeners\LogCacheMissed;
use App\Listeners\LogKeyForgotten;
use App\Listeners\LogKeyWritten;
use Illuminate\Cache\Events\CacheHit;
use Illuminate\Cache\Events\CacheMissed;
use Illuminate\Cache\Events\KeyForgotten;
use Illuminate\Cache\Events\KeyWritten;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    CacheHit::class => [
        LogCacheHit::class,
    ],
    CacheMissed::class => [
        LogCacheMissed::class,
    ],
    KeyForgotten::class => [
        LogKeyForgotten::class,
    ],
    KeyWritten::class => [
        LogKeyWritten::class,
    ],
];
```

Collections

- [Introduction](#)
 - [Creating Collections](#)
 - [Extending Collections](#)
- [Available Methods](#)
- [Higher Order Messages](#)
- [Lazy Collections](#)
 - [Introduction](#)
 - [Creating Lazy Collections](#)
 - [The Enumerable Contract](#)
 - [Lazy Collection Methods](#)

Introduction

The `Illuminate\Support\Collection` class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the `collect` helper to create a new collection instance from the array, run the `strtoupper` function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function (?string $name) {
    return strtoupper($name);
})->reject(function (string $name) {
    return empty($name);
});
```

As you can see, the `Collection` class allows you to chain its methods to perform fluent mapping and reducing of the underlying array. In general, collections are immutable, meaning every `Collection` method returns an entirely new `Collection` instance.

Creating Collections

As mentioned above, the `collect` helper returns a new `Illuminate\Support\Collection` instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]);
```



The results of `Eloquent` queries are always returned as `Collection` instances.

Extending Collections

Collections are "macroable", which allows you to add additional methods to the `Collection` class at run time. The `Illuminate\Support\Collection` class' `macro` method accepts a closure that will be executed when your macro is called. The macro closure may access the collection's other methods via `$this`, just as if it were a real method of the collection class. For example, the following code adds a `toUpper` method to the `Collection` class:

```

use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function (string $value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']

```

Typically, you should declare collection macros in the `boot` method of a [service provider](#).

Macro Arguments

If necessary, you may define macros that accept additional arguments:

```

use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;

Collection::macro('toLocale', function (string $locale) {
    return $this->map(function (string $value) use ($locale) {
        return Lang::get($value, [], $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');

```

Available Methods

For the majority of the remaining collection documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulate the underlying array. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary:

all	crossJoin	every
average	dd	except
avg	diff	filter
chunk	diffAssoc	first
chunkWhile	diffAssocUsing	firstOrFail
collapse	diffKeys	firstWhere
collect	doesntContain	flatMap
combine	dot	flatten
concat	dump	flip
contains	duplicates	forget
containsOnItem	duplicatesStrict	forPage
containsStrict	each	get
count	eachSpread	groupBy
countBy	ensure	has

hasAny	prepend	takeUntil
implode	pull	takeWhile
intersect	push	tap
intersectAssoc	put	times
intersectByKeys	random	toArray
isEmpty	range	toJson
isNotEmpty	reduce	transform
join	reduceSpread	undot
keyBy	reject	union
keys	replace	unique
last	replaceRecursive	uniqueStrict
lazy	reverse	unless
macro	search	unlessEmpty
make	select	unlessNotEmpty
map	shift	unwrap
mapInto	shuffle	value
mapSpread	skip	values
mapToGroups	skipUntil	when
mapWithKeys	skipWhile	whenEmpty
max	slice	whenNotEmpty
median	sliding	where
merge	sole	whereStrict
mergeRecursive	some	whereBetween
min	sort	whereIn
mode	sortBy	whereInStrict
nth	sortByDesc	whereInstanceOf
only	sortDesc	whereNotBetween
pad	sortKeys	whereNotIn
partition	sortKeysDesc	whereNotInStrict
percentage	sortKeysUsing	whereNotNull
pipe	splice	whereNull
pipeInto	split	wrap
pipeThrough	splitIn	zip
pluck	sum	
pop	take	

Method Listing

`all()`

The `all` method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();
// [1, 2, 3]
```

`average()`

Alias for the `avg` method.

`avg()`

The `avg` method returns the average value of a given key:

```
$average = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

chunk()

The `chunk` method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

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

This method is especially useful in [views](#) when working with a grid system such as [Bootstrap](#). For example, imagine you have a collection of [Eloquent](#) models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

chunkWhile()

The `chunkWhile` method breaks the collection into multiple, smaller collections based on the evaluation of the given callback. The `$chunk` variable passed to the closure may be used to inspect the previous element:

```
$collection = collect(str_split('AABBCCCD'));

$chunks = $collection->chunkWhile(function (string $value, int $key, Collection $chunk) {
    return $value === $chunk->last();
});

$chunks->all();

// [['A', 'A'], ['B', 'B'], ['C', 'C', 'C'], ['D']]
```

collapse()

The `collapse` method collapses a collection of arrays into a single, flat collection:

```
$collection = collect([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]);

collapsed = $collection->collapse();

collapsed->all();

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

collect()

The `collect` method returns a new `Collection` instance with the items currently in the collection:

```
$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

The `collect` method is primarily useful for converting lazy collections into standard `Collection` instances:

```
$lazyCollection = LazyCollection::make(function () {
    yield 1;
    yield 2;
    yield 3;
});

collection = $lazyCollection->collect();

collection::class;

// 'Illuminate\Support\Collection'

collection->all();

// [1, 2, 3]
```



The `collect` method is especially useful when you have an instance of `Enumerable` and need a non-lazy collection instance. Since `collect()` is part of the `Enumerable` contract, you can safely use it to get a `Collection` instance.

combine()

The `combine` method combines the values of the collection, as keys, with the values of another array or collection:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

concat()

The `concat` method appends the given `array` or collection's values onto the end of another collection:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

The `concat` method numerically reindexes keys for items concatenated onto the original collection. To maintain keys in associative collections, see the [merge](#) method.

contains()

The `contains` method determines whether the collection contains a given item. You may pass a closure to the `contains` method to determine if an element exists in the collection matching a given truth test:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function (int $value, int $key) {
    return $value > 5;
});

// false
```

Alternatively, you may pass a string to the `contains` method to determine whether the collection contains a given item value:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

You may also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

The `contains` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `containsStrict` method to filter using "strict" comparisons.

For the inverse of `contains`, see the [doesntContain](#) method.

`containsOneItem()`

The `containsOneItem` method determines whether the collection contains a single item:

```
collect([])->containsOneItem();

// false

collect(['1'])->containsOneItem();

// true

collect(['1', '2'])->containsOneItem();

// false
```

`containsStrict()`

This method has the same signature as the `contains` method; however, all values are compared using "strict" comparisons.



This method's behavior is modified when using [Eloquent Collections](#).

`count()`

The `count` method returns the total number of items in the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

`countBy()`

The `countBy` method counts the occurrences of values in the collection. By default, the method counts the occurrences of every element, allowing you to count certain "types" of elements in the collection:

```
$collection = collect([1, 2, 2, 2, 3]);  
  
$counted = $collection->countBy();  
  
$counted->all();  
  
// [1 => 1, 2 => 3, 3 => 1]
```

You pass a closure to the `countBy` method to count all items by a custom value:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);  
  
$counted = $collection->countBy(function (string $email) {  
    return substr(strrchr($email, "@"), 1);  
});  
  
$counted->all();  
  
// ['gmail.com' => 2, 'yahoo.com' => 1]
```

`crossJoin()`

The `crossJoin` method cross joins the collection's values among the given arrays or collections, returning a Cartesian product with all possible permutations:

```
$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
[
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
[
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
    [2, 'b', 'I'],
    [2, 'b', 'II'],
]
*/

```

dd()

The `dd` method dumps the collection's items and ends execution of the script:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
```

If you do not want to stop executing the script, use the `dump` method instead.

diff()

The `diff` method compares the collection against another collection or a plain PHP `array` based on its values. This method will return the values in the original collection that are not present in the given collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```



This method's behavior is modified when using [Eloquent Collections](#).

`diffAssoc()`

The `diffAssoc` method compares the collection against another collection or a plain PHP `array` based on its keys and values. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

`diffAssocUsing()`

Unlike `diffAssoc`, `diffAssocUsing` accepts a user supplied callback function for the indices comparison:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssocUsing([
    'Color' => 'yellow',
    'Type' => 'fruit',
    'Remain' => 3,
], 'strnatcasecmp');

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

The callback must be a comparison function that returns an integer less than, equal to, or greater than zero. For more information, refer to the PHP documentation on [array_diff_uassoc](#), which is the PHP function that the `diffAssocUsing` method utilizes internally.

diffKeys()

The `diffKeys` method compares the collection against another collection or a plain PHP `array` based on its keys. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

doesntContain()

The `doesntContain` method determines whether the collection does not contain a given item. You may pass a closure to the `doesntContain` method to determine if an element does not exist in the collection matching a given truth test:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->doesntContain(function (int $value, int $key) {
    return $value < 5;
});

// false
```

Alternatively, you may pass a string to the `doesntContain` method to determine whether the collection does not contain a given item value:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->doesntContain('Table');

// true

$collection->doesntContain('Desk');

// false
```

You may also pass a key / value pair to the `doesntContain` method, which will determine if the given pair does not exist in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->doesntContain('product', 'Bookcase');

// true
```

The `doesntContain` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value.

`dot()`

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

```
$collection = collect(['products' => ['desk' => ['price' => 100]]]);

$flattened = $collection->dot();

$flattened->all();

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

`dump()`

The `dump` method dumps the collection's items:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
*/
```

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

`duplicates()`

The `duplicates` method retrieves and returns duplicate values from the collection:

```
$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates();

// [2 => 'a', 4 => 'b']
```

If the collection contains arrays or objects, you can pass the key of the attributes that you wish to check for duplicate values:

```
$employees = collect([
    ['email' => 'abigail@example.com', 'position' => 'Developer'],
    ['email' => 'james@example.com', 'position' => 'Designer'],
    ['email' => 'victoria@example.com', 'position' => 'Developer'],
]);

$employees->duplicates('position');

// [2 => 'Developer']
```

`duplicatesStrict()`

This method has the same signature as the `duplicates` method; however, all values are compared using "strict" comparisons.

`each()`

The `each` method iterates over the items in the collection and passes each item to a closure:

```
$collection = collect([1, 2, 3, 4]);

$collection->each(function (int $item, int $key) {
    // ...
});
```

If you would like to stop iterating through the items, you may return `false` from your closure:

```
$collection->each(function (int $item, int $key) {
    if (/* condition */) {
        return false;
    }
});
```

eachSpread()

The `eachSpread` method iterates over the collection's items, passing each nested item value into the given callback:

```
$collection = collect(['John Doe', 35], ['Jane Doe', 33]);

$collection->eachSpread(function (string $name, int $age) {
    // ...
});
```

You may stop iterating through the items by returning `false` from the callback:

```
$collection->eachSpread(function (string $name, int $age) {
    return false;
});
```

ensure()

The `ensure` method may be used to verify that all elements of a collection are of a given type or list of types. Otherwise, an `UnexpectedValueException` will be thrown:

```
return $collection->ensure(User::class);

return $collection->ensure([User::class, Customer::class]);
```

Primitive types such as `string`, `int`, `float`, `bool`, and `array` may also be specified:

```
return $collection->ensure('int');
```



The `ensure` method does not guarantee that elements of different types will not be added to the collection at a later time.

every()

The `every` method may be used to verify that all elements of a collection pass a given truth test:

```
collect([1, 2, 3, 4])->every(function (int $value, int $key) {
    return $value > 2;
});

// false
```

If the collection is empty, the `every` method will return true:

```
$collection = collect([]);

$collection->every(function (int $value, int $key) {
    return $value > 2;
});

// true
```

except()

The `except` method returns all items in the collection except for those with the specified keys:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

For the inverse of `except`, see the [only](#) method.



This method's behavior is modified when using [Eloquent Collections](#).

filter()

The `filter` method filters the collection using the given callback, keeping only those items that pass a given truth test:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function (int $value, int $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to `false` will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

For the inverse of `filter`, see the [reject](#) method.

first()

The `first` method returns the first element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->first(function (int $value, int $key) {
    return $value > 2;
});

// 3
```

You may also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->first();

// 1
```

`firstOrFail()`

The `firstOrFail` method is identical to the `first` method; however, if no result is found, an `Illuminate\Support\ItemNotFoundException` exception will be thrown:

```
collect([1, 2, 3, 4])->firstOrFail(function (int $value, int $key) {
    return $value > 5;
};

// Throws ItemNotFoundException...
```

You may also call the `firstOrFail` method with no arguments to get the first element in the collection. If the collection is empty, an `Illuminate\Support\ItemNotFoundException` exception will be thrown:

```
collect([])->firstOrFail();

// Throws ItemNotFoundException...
```

`firstWhere()`

The `firstWhere` method returns the first element in the collection with the given key / value pair:

```
$collection = collect([
    ['name' => 'Regena', 'age' => null],
    ['name' => 'Linda', 'age' => 14],
    ['name' => 'Diego', 'age' => 23],
    ['name' => 'Linda', 'age' => 84],
]);

$collection->firstWhere('name', 'Linda');

// ['name' => 'Linda', 'age' => 14]
```

You may also call the `firstWhere` method with a comparison operator:

```
$collection->firstWhere('age', '>=', 18);

// ['name' => 'Diego', 'age' => 23]
```

Like the `where` method, you may pass one argument to the `firstWhere` method. In this scenario, the `firstWhere` method will return the first item where the given item key's value is "truthy":

```
$collection->firstWhere('age');

// ['name' => 'Linda', 'age' => 14]
```

`flatMap()`

The `flatMap` method iterates through the collection and passes each value to the given closure. The closure is free to modify the item and return it, thus forming a new collection of modified items. Then, the array is flattened by one level:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function (array $values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

`flatten()`

The `flatten` method flattens a multi-dimensional collection into a single dimension:

```
$collection = collect([
    'name' => 'taylor',
    'languages' => [
        'php', 'javascript'
    ]
]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

If necessary, you may pass the `flatten` method a "depth" argument:

```
$collection = collect([
    'Apple' => [
        [
            'name' => 'iPhone 6S',
            'brand' => 'Apple'
        ],
    ],
    'Samsung' => [
        [
            'name' => 'Galaxy S7',
            'brand' => 'Samsung'
        ],
    ],
]);
$products = $collection->flatten(1);

$products->values()->all();

/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
*/

```

In this example, calling `flatten` without providing the depth would have also flattened the nested arrays, resulting in `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Providing a depth allows you to specify the number of levels nested arrays will be flattened.

`flip()`

The `flip` method swaps the collection's keys with their corresponding values:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

`forget()`

The `forget` method removes an item from the collection by its key:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```



Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

forPage()

The `forPage` method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

get()

The `get` method returns the item at a given key. If the key does not exist, `null` is returned:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

You may optionally pass a default value as the second argument:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('age', 34);

// 34
```

You may even pass a callback as the method's default value. The result of the callback will be returned if the specified key does not exist:

```
$collection->get('email', function () {
    return 'taylor@example.com';
});

// taylor@example.com
```

groupBy()

The `groupBy` method groups the collection's items by a given key:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->all();

/*
[
    'account-x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

Instead of passing a string `[key]`, you may pass a callback. The callback should return the value you wish to key the group by:

```
$grouped = $collection->groupBy(function (array $item, int $key) {
    return substr($item['account_id'], -3);
});

$grouped->all();

/*
[
    'x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

Multiple grouping criteria may be passed as an array. Each array element will be applied to the corresponding level within a multi-dimensional array:

```

$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy(['skill'], function (array $item) {
    return $item['roles'];
}), preserveKeys: true);

/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
];
*/

```

has()

The `has` method determines if a given key exists in the collection:

```

$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->has('product');

// true

$collection->has(['product', 'amount']);

// true

$collection->has(['amount', 'price']);

// false

```

hasAny()

The `hasAny` method determines whether any of the given keys exist in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->hasAny(['product', 'price']);

// true

$collection->hasAny(['name', 'price']);

// false
```

implode()

The `implode` method joins items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

If the collection contains simple strings or numeric values, you should pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

You may pass a closure to the `implode` method if you would like to format the values being implode:

```
$collection->implode(function (array $item, int $key) {
    return strtoupper($item['product']);
}, ', ');

// DESK, CHAIR
```

intersect()

The `intersect` method removes any values from the original collection that are not present in the given `array` or collection. The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```



This method's behavior is modified when using [Eloquent Collections](#).

intersectAssoc()

The `intersectAssoc` method compares the original collection against another collection or `array`, returning the key / value pairs that are present in all of the given collections:

```
$collection = collect([
    'color' => 'red',
    'size' => 'M',
    'material' => 'cotton'
]);

$intersect = $collection->intersectAssoc([
    'color' => 'blue',
    'size' => 'M',
    'material' => 'polyester'
]);

$intersect->all();

// ['size' => 'M']
```

intersectByKeys()

The `intersectByKeys` method removes any keys and their corresponding values from the original collection that are not present in the given `array` or collection:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009,
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011,
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

isEmpty()

The `isEmpty` method returns `true` if the collection is empty; otherwise, `false` is returned:

```
collect([])->isEmpty();

// true
```

isNotEmpty()

The `isNotEmpty` method returns `true` if the collection is not empty; otherwise, `false` is returned:

```
collect([])->isNotEmpty();
// false
```

join()

The `join` method joins the collection's values with a string. Using this method's second argument, you may also specify how the final element should be appended to the string:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'
collect(['a', 'b', 'c'])->join(', ', ' and '); // 'a, b, and c'
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'
collect(['a'])->join(', ', ' and '); // 'a'
collect([])->join(', ', ' and '); // ''
```

keyBy()

The `keyBy` method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function (array $item, int $key) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
```

keys()

The `keys` method returns all of the collection's keys:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$keys = $collection->keys();
$keys->all();
// ['prod-100', 'prod-200']
```

last()

The `last` method returns the last element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->last(function (int $value, int $key) {
    return $value < 3;
});
// 2
```

You may also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->last();
// 4
```

lazy()

The `lazy` method returns a new `LazyCollection` instance from the underlying array of items:

```
$lazyCollection = collect([1, 2, 3, 4])->lazy();
$lazyCollection::class;
// Illuminate\Support\LazyCollection
$lazyCollection->all();
// [1, 2, 3, 4]
```

This is especially useful when you need to perform transformations on a huge `Collection` that contains many items:

```
$count = $hugeCollection
->lazy()
->where('country', 'FR')
->where('balance', '>', '100')
->count();
```

By converting the collection to a `LazyCollection`, we avoid having to allocate a ton of additional memory. Though the original collection still keeps *its* values in memory, the subsequent filters will not. Therefore, virtually no additional memory will be allocated when filtering the collection's results.

macro()

The static `macro` method allows you to add methods to the `Collection` class at run time. Refer to the documentation on [extending collections](#) for more information.

make()

The static `make` method creates a new collection instance. See the [Creating Collections](#) section.

map()

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function (int $item, int $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```



Like most other collection methods, `map` returns a new collection instance; it does not modify the collection it is called on. If you want to transform the original collection, use the [transform](#) method.

mapInto()

The `mapInto()` method iterates over the collection, creating a new instance of the given class by passing the value into the constructor:

```
class Currency
{
    /**
     * Create a new currency instance.
     */
    function __construct(
        public string $code
    ) {}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

mapSpread()

The `mapSpread` method iterates over the collection's items, passing each nested item value into the given closure. The closure is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function (int $even, int $odd) {
    return $even + $odd;
});

$sequence->all();

// [1, 5, 9, 13, 17]
```

`mapToGroups()`

The `mapToGroups` method groups the collection's items by the given closure. The closure should return an associative array containing a single key / value pair, thus forming a new collection of grouped values:

```
$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function (array $item, int $key) {
    return [$item['department'] => $item['name']];
});

$grouped->all();

/*
[
    'Sales' => ['John Doe', 'Jane Doe'],
    'Marketing' => ['Johnny Doe'],
]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

`mapWithKeys()`

The `mapWithKeys` method iterates through the collection and passes each value to the given callback. The callback should return an associative array containing a single key / value pair:

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
]);

$keyed = $collection->mapWithKeys(function (array $item, int $key) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/
```

max()

The `max` method returns the maximum value of a given key:

```
$max = collect([
    ['foo' => 10],
    ['foo' => 20]
])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

median()

The `median` method returns the median value of a given key:

```
$median = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

merge()

The `merge` method merges the given array or collection with the original collection. If a string key in the given items matches a string key in the original collection, the given item's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given item's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

mergeRecursive()

The `mergeRecursive` method merges the given array or collection recursively with the original collection. If a string key in the given items matches a string key in the original collection, then the values for these keys are merged together into an array, and this is done recursively:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive([
    'product_id' => 2,
    'price' => 200,
    'discount' => false
]);

$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

min()

The `min` method returns the minimum value of a given key:

```
$min = collect(['foo' => 10, 'foo' => 20])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

mode()

The `mode` method returns the mode value of a given key:

```
$mode = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]

$mode = collect([1, 1, 2, 2])->mode();

// [1, 2]
```

nth()

The `nth` method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

You may optionally pass a starting offset as the second argument:

```
$collection->nth(4, 1);

// ['b', 'f']
```

only()

The `only` method returns the items in the collection with the specified keys:

```
$collection = collect([
    'product_id' => 1,
    'name' => 'Desk',
    'price' => 100,
    'discount' => false
]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

For the inverse of `only`, see the [except](#) method.



This method's behavior is modified when using [Eloquent Collections](#).

`pad()`

The `pad` method will fill the array with the given value until the array reaches the specified size. This method behaves like the [array_pad](#) PHP function.

To pad to the left, you should specify a negative size. No padding will take place if the absolute value of the given size is less than or equal to the length of the array:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

`partition()`

The `partition` method may be combined with PHP array destructuring to separate elements that pass a given truth test from those that do not:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

[$underThree, $equalOrAboveThree] = $collection->partition(function (int $i) {
    return $i < 3;
});

$underThree->all();
// [1, 2]

$equalOrAboveThree->all();
// [3, 4, 5, 6]
```

percentage()

The `percentage` method may be used to quickly determine the percentage of items in the collection that pass a given truth test:

```
$collection = collect([1, 1, 2, 2, 2, 3]);

$percentage = $collection->percentage(fn ($value) => $value === 1);

// 33.33
```

By default, the percentage will be rounded to two decimal places. However, you may customize this behavior by providing a second argument to the method:

```
$percentage = $collection->percentage(fn ($value) => $value === 1, precision: 3);

// 33.333
```

pipe()

The `pipe` method passes the collection to the given closure and returns the result of the executed closure:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function (Collection $collection) {
    return $collection->sum();
});

// 6
```

pipeInto()

The `pipeInto` method creates a new instance of the given class and passes the collection into the constructor:

```
class ResourceCollection
{
    /**
     * Create a new ResourceCollection instance.
     */
    public function __construct(
        public Collection $collection,
    ) {}

$collection = collect([1, 2, 3]);

$resource = $collection->pipeInto(ResourceCollection::class);

$resource->collection->all();

// [1, 2, 3]
```

pipeThrough()

The `pipeThrough` method passes the collection to the given array of closures and returns the result of the executed closures:

```
use Illuminate\Support\Collection;

$collection = collect([1, 2, 3]);

$result = $collection->pipeThrough([
    function (Collection $collection) {
        return $collection->merge([4, 5]);
    },
    function (Collection $collection) {
        return $collection->sum();
    },
]);
// 15
```

pluck()

The `pluck` method retrieves all of the values for a given key:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

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

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

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

The `pluck` method also supports retrieving nested values using "dot" notation:

```
$collection = collect([
    [
        'name' => 'Laracon',
        'speakers' => [
            'first_day' => ['Rosa', 'Judith'],
        ],
    ],
    [
        'name' => 'VueConf',
        'speakers' => [
            'first_day' => ['Abigail', 'Joey'],
        ],
    ],
]);
$plucked = $collection->pluck('speakers.first_day');

$plucked->all();

// [['Rosa', 'Judith'], ['Abigail', 'Joey']]
```

If duplicate keys exist, the last matching element will be inserted into the plucked collection:

```
$collection = collect([
    ['brand' => 'Tesla', 'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla', 'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

`pop()`

The `pop` method removes and returns the last item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

You may pass an integer to the `pop` method to remove and return multiple items from the end of a collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop(3);

// collect([5, 4, 3])

$collection->all();

// [1, 2]
```

`prepend()`

The `prepend` method adds an item to the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to specify the key of the prepended item:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

`pull()`

The `pull` method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

push()

The `push` method appends an item to the end of the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

put()

The `put` method sets the given key and value in the collection:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

The `random` method returns a random item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

You may pass an integer to `random` to specify how many items you would like to randomly retrieve. A collection of items is always returned when explicitly passing the number of items you wish to receive:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

If the collection instance has fewer items than requested, the `random` method will throw an `InvalidArgumentException`.

The `random` method also accepts a closure, which will receive the current collection instance:

```
use Illuminate\Support\Collection;

$random = $collection->random(fn (Collection $items) => min(10, count($items)));

$random->all();

// [1, 2, 3, 4, 5] - (retrieved randomly)
```

`range()`

The `range` method returns a collection containing integers between the specified range:

```
$collection = collect()->range(3, 6);

$collection->all();

// [3, 4, 5, 6]
```

`reduce()`

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function (?int $carry, int $item) {
    return $carry + $item;
});

// 6
```

The value for `$carry` on the first iteration is `null`; however, you may specify its initial value by passing a second argument to `reduce`:

```
$collection->reduce(function (int $carry, int $item) {
    return $carry + $item;
}, 4);

// 10
```

The `reduce` method also passes array keys in associative collections to the given callback:

```
$collection = collect([
    'usd' => 1400,
    'gbp' => 1200,
    'eur' => 1000,
]);

$ratio = [
    'usd' => 1,
    'gbp' => 1.37,
    'eur' => 1.22,
];

$collection->reduce(function (int $carry, int $value, int $key) use ($ratio) {
    return $carry + ($value * $ratio[$key]);
});

// 4264
```

reduceSpread()

The `reduceSpread` method reduces the collection to an array of values, passing the results of each iteration into the subsequent iteration. This method is similar to the `reduce` method; however, it can accept multiple initial values:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
->get()
->reduceSpread(function (int $creditsRemaining, Collection $batch, Image $image) {
    if ($creditsRemaining >= $image->creditsRequired()) {
        $batch->push($image);

        $creditsRemaining -= $image->creditsRequired();
    }

    return [$creditsRemaining, $batch];
}, $creditsAvailable, collect());
```

reject()

The `reject` method filters the collection using the given closure. The closure should return `true` if the item should be removed from the resulting collection:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function (int $value, int $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

For the inverse of the `reject` method, see the `filter` method.

replace()

The `replace` method behaves similarly to `merge`; however, in addition to overwriting matching items that have string keys, the `replace` method will also overwrite items in the collection that have matching numeric keys:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all();

// ['Taylor', 'Victoria', 'James', 'Finn']
```

`replaceRecursive()`

This method works like `replace`, but it will recur into arrays and apply the same replacement process to the inner values:

```
$collection = collect([
    'Taylor',
    'Abigail',
    [
        'James',
        'Victoria',
        'Finn'
    ]
]);

$replaced = $collection->replaceRecursive([
    'Charlie',
    2 => [1 => 'King']
]);

$replaced->all();

// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

`reverse()`

The `reverse` method reverses the order of the collection's items, preserving the original keys:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
[
    4 => 'e',
    3 => 'd',
    2 => 'c',
    1 => 'b',
    0 => 'a',
]
```

`search()`

The `search` method searches the collection for the given value and returns its key if found. If the item is not found, `false` is returned:

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value. To use "strict" comparison, pass `true` as the second argument to the method:

```
collect([2, 4, 6, 8])->search('4', $strict = true);

// false
```

Alternatively, you may provide your own closure to search for the first item that passes a given truth test:

```
collect([2, 4, 6, 8])->search(function (int $item, int $key) {
    return $item > 5;
});

// 2
```

`select()`

The `select` method selects the given keys from the collection, similar to an SQL `SELECT` statement:

```
$users = collect([
    ['name' => 'Taylor Otwell', 'role' => 'Developer', 'status' => 'active'],
    ['name' => 'Victoria Faith', 'role' => 'Researcher', 'status' => 'active'],
]);

$users->select(['name', 'role']);

/*
[
    ['name' => 'Taylor Otwell', 'role' => 'Developer'],
    ['name' => 'Victoria Faith', 'role' => 'Researcher'],
],
*/
```

`shift()`

The `shift` method removes and returns the first item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

You may pass an integer to the `shift` method to remove and return multiple items from the beginning of a collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift(3);

// collect([1, 2, 3])

$collection->all();

// [4, 5]
```

`shuffle()`

The `shuffle` method randomly shuffles the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

`skip()`

The `skip` method returns a new collection, with the given number of elements removed from the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);

$collection->all();

// [5, 6, 7, 8, 9, 10]
```

`skipUntil()`

The `skipUntil` method skips over items from the collection until the given callback returns `true` and then returns the remaining items in the collection as a new collection instance:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(function (int $item) {
    return $item >= 3;
});

$subset->all();

// [3, 4]
```

You may also pass a simple value to the `skipUntil` method to skip all items until the given value is found:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(3);

$subset->all();

// [3, 4]
```



If the given value is not found or the callback never returns `true`, the `skipUntil` method will return an empty collection.

`skipWhile()`

The `skipWhile` method skips over items from the collection while the given callback returns `true` and then returns the remaining items in the collection as a new collection:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipWhile(function (int $item) {
    return $item <= 3;
});

$subset->all();

// [4]
```



If the callback never returns `false`, the `skipWhile` method will return an empty collection.

`slice()`

The `slice` method returns a slice of the collection starting at the given index:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

The returned slice will preserve keys by default. If you do not wish to preserve the original keys, you can use the [values](#) method to reindex them.

sliding()

The `sliding` method returns a new collection of chunks representing a "sliding window" view of the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(2);

$chunks->toArray();

// [[1, 2], [2, 3], [3, 4], [4, 5]]
```

This is especially useful in conjunction with the `eachSpread` method:

```
$transactions->sliding(2)->eachSpread(function (Collection $previous, Collection $current) {
    $current->total = $previous->total + $current->amount;
});
```

You may optionally pass a second "step" value, which determines the distance between the first item of every chunk:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(3, step: 2);

$chunks->toArray();

// [[1, 2, 3], [3, 4, 5]]
```

sole()

The `sole` method returns the first element in the collection that passes a given truth test, but only if the truth test matches exactly one element:

```
collect([1, 2, 3, 4])->sole(function (int $value, int $key) {
    return $value === 2;
});

// 2
```

You may also pass a key / value pair to the `sole` method, which will return the first element in the collection that matches the given pair, but only if it exactly one element matches:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->sole('product', 'Chair');

// ['product' => 'Chair', 'price' => 100]
```

Alternatively, you may also call the `sole` method with no argument to get the first element in the collection if there is only one element:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
]);

$collection->sole();

// ['product' => 'Desk', 'price' => 200]
```

If there are no elements in the collection that should be returned by the `sole` method, an `\Illuminate\Collections\ItemNotFoundException` exception will be thrown. If there is more than one element that should be returned, an `\Illuminate\Collections\MultipleItemsFoundException` will be thrown.

`some()`

Alias for the `contains` method.

`sort()`

The `sort` method sorts the collection. The sorted collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

If your sorting needs are more advanced, you may pass a callback to `sort` with your own algorithm. Refer to the PHP documentation on `uasort`, which is what the collection's `sort` method calls utilizes internally.



If you need to sort a collection of nested arrays or objects, see the [sortBy](#) and [sortByDesc](#) methods.

sortBy()

The `sortBy` method sorts the collection by the given key. The sorted collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/
```

The `sortBy` method accepts sort flags as its second argument:

```
$collection = collect([
    ['title' => 'Item 1'],
    ['title' => 'Item 12'],
    ['title' => 'Item 3'],
]);

$sorted = $collection->sortBy('title', SORT_NATURAL);

$sorted->values()->all();

/*
[
    ['title' => 'Item 1'],
    ['title' => 'Item 3'],
    ['title' => 'Item 12'],
]
*/
```

Alternatively, you may pass your own closure to determine how to sort the collection's values:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
$sorted = $collection->sortBy(function (array $product, int $key) {
    return count($product['colors']);
});
$sorted->values()->all();
/*
[
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]
*/

```

If you would like to sort your collection by multiple attributes, you may pass an array of the attributes that you wish to sort by:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);
$sorted = $collection->sortBy(['name', 'age']);
$sorted->values()->all();
/*
[
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Taylor Otwell', 'age' => 36],
]
*/

```

When sorting in multiple attributes and directions, you may pass an array of sort operations to the `sortBy` method. Each sort operation should be an array consisting of the attribute that you wish to sort by and the direction of the desired sort:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    ['name', 'asc'],
    ['age', 'desc'],
]);

$sorted->values()->all();

/*
[
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Taylor Otwell', 'age' => 34],
]
*/

```

When sorting a collection by multiple attributes, you may also provide closures that define each sort operation:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    fn (array $a, array $b) => $a['name'] <=> $b['name'],
    fn (array $a, array $b) => $b['age'] <=> $a['age'],
]);

$sorted->values()->all();

/*
[
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Taylor Otwell', 'age' => 34],
]
*/

```

sortByDesc()

This method has the same signature as the [sortBy](#) method, but will sort the collection in the opposite order.

sortDesc()

This method will sort the collection in the opposite order as the [sort](#) method:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sortDesc();

$sorted->values()->all();

// [5, 4, 3, 2, 1]
```

Unlike `sort`, you may not pass a closure to `sortDesc`. Instead, you should use the `sort` method and invert your comparison.

`sortKeys()`

The `sortKeys` method sorts the collection by the keys of the underlying associative array:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
[
    'first' => 'John',
    'id' => 22345,
    'last' => 'Doe',
]
*/
```

`sortKeysDesc()`

This method has the same signature as the `sortKeys` method, but will sort the collection in the opposite order.

`sortKeysUsing()`

The `sortKeysUsing` method sorts the collection by the keys of the underlying associative array using a callback:

```
$collection = collect([
    'ID' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);
$sorted = $collection->sortKeysUsing('strnatcasecmp');
$sorted->all();
/*
[
    'first' => 'John',
    'ID' => 22345,
    'last' => 'Doe',
]
*/
```

The callback must be a comparison function that returns an integer less than, equal to, or greater than zero. For more information, refer to the PHP documentation on [uksort](#), which is the PHP function that `sortKeysUsing` method utilizes internally.

`splice()`

The `splice` method removes and returns a slice of items starting at the specified index:

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2);
$chunk->all();
// [3, 4, 5]
$collection->all();
// [1, 2]
```

You may pass a second argument to limit the size of the resulting collection:

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2, 1);
$chunk->all();
// [3]
$collection->all();
// [1, 2, 4, 5]
```

In addition, you may pass a third argument containing the new items to replace the items removed from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

split()

The `split` method breaks a collection into the given number of groups:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->all();

// [[1, 2], [3, 4], [5]]
```

splitIn()

The `splitIn` method breaks a collection into the given number of groups, filling non-terminal groups completely before allocating the remainder to the final group:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$groups = $collection->splitIn(3);

$groups->all();

// [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

sum()

The `sum` method returns the sum of all items in the collection:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

If the collection contains nested arrays or objects, you should pass a key that will be used to determine which values to sum:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

In addition, you may pass your own closure to determine which values of the collection to sum:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function (array $product) {
    return count($product['colors']);
});

// 6
```

`take()`

The `take` method returns a new collection with the specified number of items:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

You may also pass a negative integer to take the specified number of items from the end of the collection:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

`takeUntil()`

The `takeUntil` method returns items in the collection until the given callback returns `true`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(function (int $item) {
    return $item >= 3;
});

$subset->all();

// [1, 2]
```

You may also pass a simple value to the `takeUntil` method to get the items until the given value is found:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(3);

$subset->all();

// [1, 2]
```



If the given value is not found or the callback never returns `true`, the `takeUntil` method will return all items in the collection.

`takeWhile()`

The `takeWhile` method returns items in the collection until the given callback returns `false`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeWhile(function (int $item) {
    return $item < 3;
});

$subset->all();

// [1, 2]
```



If the callback never returns `false`, the `takeWhile` method will return all items in the collection.

`tap()`

The `tap` method passes the collection to the given callback, allowing you to "tap" into the collection at a specific point and do something with the items while not affecting the collection itself. The collection is then returned by the `tap` method:

```
collect([2, 4, 3, 1, 5])
->sort()
->tap(function (Collection $collection) {
    Log::debug('Values after sorting', $collection->values()-->all());
})
->shift();

// 1
```

times()

The static `times` method creates a new collection by invoking the given closure a specified number of times:

```
$collection = Collection::times(10, function (int $number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

toArray()

The `toArray` method converts the collection into a plain PHP `array`. If the collection's values are [Eloquent](#) models, the models will also be converted to arrays:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

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



`toArray` also converts all of the collection's nested objects that are an instance of `Arrayable` to an array. If you want to get the raw array underlying the collection, use the `all` method instead.

toJson()

The `toJson` method converts the collection into a JSON serialized string:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name": "Desk", "price": 200}'
```

transform()

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function (int $item, int $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```



Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the `map` method.

undot()

The `undot` method expands a single-dimensional collection that uses "dot" notation into a multi-dimensional collection:

```
$person = collect([
    'name.first_name' => 'Marie',
    'name.last_name' => 'Valentine',
    'address.line_1' => '2992 Eagle Drive',
    'address.line_2' => '',
    'address.suburb' => 'Detroit',
    'address.state' => 'MI',
    'address.postcode' => '48219'
]);

$person = $person->undot();

$person->toArray();

/*
[
    "name" => [
        "first_name" => "Marie",
        "last_name" => "Valentine",
    ],
    "address" => [
        "line_1" => "2992 Eagle Drive",
        "line_2" => "",
        "suburb" => "Detroit",
        "state" => "MI",
        "postcode" => "48219",
    ],
]
```

union()

The `union` method adds the given array to the collection. If the given array contains keys that are already in the original collection, the original collection's values will be preferred:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['d']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

unique()

The `unique` method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

When dealing with nested arrays or objects, you may specify the key used to determine uniqueness:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
]
*/
```

Finally, you may also pass your own closure to the `unique` method to specify which value should determine an item's uniqueness:

```
$unique = $collection->unique(function (array $item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]
*/

```

The `unique` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `uniqueStrict` method to filter using "strict" comparisons.



This method's behavior is modified when using [Eloquent Collections](#).

`uniqueStrict()`

This method has the same signature as the `unique` method; however, all values are compared using "strict" comparisons.

`unless()`

The `unless` method will execute the given callback unless the first argument given to the method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function (Collection $collection) {
    return $collection->push(4);
});

$collection->unless(false, function (Collection $collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

A second callback may be passed to the `unless` method. The second callback will be executed when the first argument given to the `unless` method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function (Collection $collection) {
    return $collection->push(4);
}, function (Collection $collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

For the inverse of `unless`, see the [when](#) method.

`unlessEmpty()`

Alias for the [whenNotEmpty](#) method.

`unlessNotEmpty()`

Alias for the [whenEmpty](#) method.

`unwrap()`

The static `unwrap` method returns the collection's underlying items from the given value when applicable:

```
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

`value()`

The `value` method retrieves a given value from the first element of the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Speaker', 'price' => 400],
]);

$value = $collection->value('price');

// 200
```

`values()`

The `values` method returns a new collection with the keys reset to consecutive integers:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200],
]);

$values = $collection->values();

$values->all();

/*
[
    0 => ['product' => 'Desk', 'price' => 200],
    1 => ['product' => 'Desk', 'price' => 200],
]
*/
```

when()

The `when` method will execute the given callback when the first argument given to the method evaluates to `true`. The collection instance and the first argument given to the `when` method will be provided to the closure:

```
$collection = collect([1, 2, 3]);

$collection->when(true, function (Collection $collection, int $value) {
    return $collection->push(4);
});

$collection->when(false, function (Collection $collection, int $value) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]
```

A second callback may be passed to the `when` method. The second callback will be executed when the first argument given to the `when` method evaluates to `false`:

```
$collection = collect([1, 2, 3]);

$collection->when(false, function (Collection $collection, int $value) {
    return $collection->push(4);
}, function (Collection $collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

For the inverse of `when`, see the `unless` method.

whenEmpty()

The `whenEmpty` method will execute the given callback when the collection is empty:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function (Collection $collection) {
    return $collection->push('Adam');
});

$collection->all();

// ['Michael', 'Tom']

$collection = collect();

$collection->whenEmpty(function (Collection $collection) {
    return $collection->push('Adam');
});

$collection->all();

// ['Adam']
```

A second closure may be passed to the `whenEmpty` method that will be executed when the collection is not empty:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function (Collection $collection) {
    return $collection->push('Adam');
}, function (Collection $collection) {
    return $collection->push('Taylor');
});

$collection->all();

// ['Michael', 'Tom', 'Taylor']
```

For the inverse of `whenEmpty`, see the `whenNotEmpty` method.

`whenNotEmpty()`

The `whenNotEmpty` method will execute the given callback when the collection is not empty:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']

$collection = collect();

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
});

$collection->all();

// []
```

A second closure may be passed to the `whenNotEmpty` method that will be executed when the collection is empty:

```
$collection = collect();

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
}, function (Collection $collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

For the inverse of `whenNotEmpty`, see the `whenEmpty` method.

`where()`

The `where` method filters the collection by a given key / value pair:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);
$filtered = $collection->where('price', 100);
$filtered->all();
/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

The `where` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereStrict` method to filter using "strict" comparisons.

Optionally, you may pass a comparison operator as the second parameter. Supported operators are: '`==`', '`!=`', '`!=`', '`==`', '`=`', '`<>`', '`>`', '`<`', '`>=`', and '`<=`':

```
$collection = collect([
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ['name' => 'Sue', 'deleted_at' => null],
]);
$filtered = $collection->where('deleted_at', '!=', null);
$filtered->all();
/*
[
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
]
*/
```

`whereStrict()`

This method has the same signature as the `where` method; however, all values are compared using "strict" comparisons.

`whereBetween()`

The `whereBetween` method filters the collection by determining if a specified item value is within a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);
$filtered = $collection->whereBetween('price', [100, 200]);
$filtered->all();
/*
[
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]
*/

```

whereIn()

The `whereIn` method removes elements from the collection that do not have a specified item value that is contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);
$filtered = $collection->whereIn('price', [150, 200]);
$filtered->all();
/*
[
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
]
*/

```

The `whereIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereInStrict` method to filter using "strict" comparisons.

whereInStrict()

This method has the same signature as the `whereIn` method; however, all values are compared using "strict" comparisons.

whereInstanceOf()

The `whereInstanceOf` method filters the collection by a given class type:

```
use App\Models\User;
use App\Models\Post;

$collection = collect([
    new User,
    new User,
    new Post,
]);

$filtered = $collection->whereInstanceOf(User::class);

$filtered->all();

// [App\Models\User, App\Models\User]
```

whereNotBetween()

The `whereNotBetween` method filters the collection by determining if a specified item value is outside of a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Pencil', 'price' => 30],
]
*/
```

whereNotIn()

The `whereNotIn` method removes elements from the collection that have a specified item value that is contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);
$filtered = $collection->whereNotIn('price', [150, 200]);
$filtered->all();
/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

The `whereNotIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereNotInStrict` method to filter using "strict" comparisons.

`whereNotInStrict()`

This method has the same signature as the `whereNotIn` method; however, all values are compared using "strict" comparisons.

`whereNotNull()`

The `whereNotNull` method returns items from the collection where the given key is not `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);
$filtered = $collection->whereNotNull('name');
$filtered->all();
/*
[
    ['name' => 'Desk'],
    ['name' => 'Bookcase'],
]
*/
```

`whereNull()`

The `whereNull` method returns items from the collection where the given key is `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);
$filtered = $collection->whereNull('name');
$filtered->all();
/*
[
    ['name' => null],
]
*/
```

wrap()

The static `wrap` method wraps the given value in a collection when applicable:

```
use Illuminate\Support\Collection;

$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

zip()

The `zip` method merges together the values of the given array with the values of the original collection at their corresponding index:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Higher Order Messages

Collections also provide support for "higher order messages", which are short-cuts for performing common actions on collections. The collection methods that provide higher order messages are: `average`, `avg`, `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `groupBy`, `keyBy`, `map`, `max`, `min`, `partition`, `reject`, `skipUntil`, `skipWhile`, `some`, `sortBy`, `sortByDesc`, `sum`, `takeUntil`, `takeWhile`, and `unique`.

Each higher order message can be accessed as a dynamic property on a collection instance. For instance, let's use the `each` higher order message to call a method on each object within a collection:

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Likewise, we can use the `sum` higher order message to gather the total number of "votes" for a collection of users:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Lazy Collections

Introduction



Before learning more about Laravel's lazy collections, take some time to familiarize yourself with [PHP generators](#).

To supplement the already powerful `Collection` class, the `LazyCollection` class leverages PHP's [generators](#) to allow you to work with very large datasets while keeping memory usage low.

For example, imagine your application needs to process a multi-gigabyte log file while taking advantage of Laravel's collection methods to parse the logs. Instead of reading the entire file into memory at once, lazy collections may be used to keep only a small part of the file in memory at a given time:

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function (array $lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Or, imagine you need to iterate through 10,000 Eloquent models. When using traditional Laravel collections, all 10,000 Eloquent models must be loaded into memory at the same time:

```
use App\Models\User;

$users = User::all()->filter(function (User $user) {
    return $user->id > 500;
});
```

However, the query builder's `cursor` method returns a `LazyCollection` instance. This allows you to still only run a single query against the database but also only keep one Eloquent model loaded in memory at a time. In this example, the `filter` callback is not executed until we actually iterate over each user individually, allowing for a drastic reduction in memory usage:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Creating Lazy Collections

To create a lazy collection instance, you should pass a PHP generator function to the collection's `make` method:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
});
```

The Enumerable Contract

Almost all methods available on the `Collection` class are also available on the `LazyCollection` class. Both of these classes implement the `Illuminate\Support\Enumerable` contract, which defines the following methods:

<code>all</code>	<code>dd</code>	<code>firstWhere</code>
<code>average</code>	<code>diff</code>	<code>flatMap</code>
<code>avg</code>	<code>diffAssoc</code>	<code>flatten</code>
<code>chunk</code>	<code>diffKeys</code>	<code>flip</code>
<code>chunkWhile</code>	<code>dump</code>	<code>forPage</code>
<code>collapse</code>	<code>duplicates</code>	<code>get</code>
<code>collect</code>	<code>duplicatesStrict</code>	<code>groupBy</code>
<code>combine</code>	<code>each</code>	<code>has</code>
<code>concat</code>	<code>eachSpread</code>	<code>implode</code>
<code>contains</code>	<code>every</code>	<code>intersect</code>
<code>containsStrict</code>	<code>except</code>	<code>intersectAssoc</code>
<code>count</code>	<code>filter</code>	<code>intersectByKeys</code>
<code>countBy</code>	<code>first</code>	<code>isEmpty</code>
<code>crossJoin</code>	<code>firstOrFail</code>	<code>isNotEmpty</code>

join	random	toJson
keyBy	reduce	union
keys	reject	unique
last	replace	uniqueStrict
macro	replaceRecursive	unless
make	reverse	unlessEmpty
map	search	unlessNotEmpty
mapInto	shuffle	unwrap
mapSpread	skip	values
mapToGroups	slice	when
mapWithKeys	sole	whenEmpty
max	some	whenNotEmpty
median	sort	where
merge	sortBy	whereStrict
mergeRecursive	sortByDesc	whereBetween
min	sortKeys	whereIn
mode	sortKeysDesc	whereInStrict
nth	split	whereInstanceOf
only	sum	whereNotBetween
pad	take	whereNotIn
partition	tap	whereNotInStrict
pipe	times	wrap
pluck	toArray	zip



Methods that mutate the collection (such as `shift`, `pop`, `prepend` etc.) are **not** available on the `LazyCollection` class.

Lazy Collection Methods

In addition to the methods defined in the `Enumerable` contract, the `LazyCollection` class contains the following methods:

`takeUntilTimeout()`

The `takeUntilTimeout` method returns a new lazy collection that will enumerate values until the specified time. After that time, the collection will then stop enumerating:

```
$lazyCollection = LazyCollection::times(INF)
->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function (int $number) {
    dump($number);

    sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

To illustrate the usage of this method, imagine an application that submits invoices from the database using a cursor. You could define a [scheduled task](#) that runs every 15 minutes and only processes invoices for a maximum of 14 minutes:

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
    ->takeUntilTimeout(
        Carbon::createFromTimestamp(Laravel_START)->add(14, 'minutes')
    )
    ->each(fn (Invoice $invoice) => $invoice->submit());
```

tapEach()

While the `each` method calls the given callback for each item in the collection right away, the `tapEach` method only calls the given callback as the items are being pulled out of the list one by one:

```
// Nothing has been dumped so far...
$lazyCollection = LazyCollection::times(INF)->tapEach(function (int $value) {
    dump($value);
});

// Three items are dumped...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

remember()

The `remember` method returns a new lazy collection that will remember any values that have already been enumerated and will not retrieve them again on subsequent collection enumerations:

```
// No query has been executed yet...
$users = User::cursor()->remember();

// The query is executed...
// The first 5 users are hydrated from the database...
$users->take(5)->all();

// First 5 users come from the collection's cache...
// The rest are hydrated from the database...
$users->take(20)->all();
```

Contracts

- [Introduction](#)
 - [Contracts vs. Facades](#)
- [When to Use Contracts](#)
- [How to Use Contracts](#)
- [Contract Reference](#)

Introduction

Laravel's "contracts" are a set of interfaces that define the core services provided by the framework. For example, an `Illuminate\Contracts\Queue\Queue` contract defines the methods needed for queueing jobs, while the `Illuminate\Contracts\Mail\Mailer` contract defines the methods needed for sending e-mail.

Each contract has a corresponding implementation provided by the framework. For example, Laravel provides a queue implementation with a variety of drivers, and a mailer implementation that is powered by [Symfony Mailer](#).

All of the Laravel contracts live in [their own GitHub repository](#). This provides a quick reference point for all available contracts, as well as a single, decoupled package that may be utilized when building packages that interact with Laravel services.

Contracts vs. Facades

Laravel's [facades](#) and helper functions provide a simple way of utilizing Laravel's services without needing to type-hint and resolve contracts out of the service container. In most cases, each facade has an equivalent contract.

Unlike facades, which do not require you to require them in your class' constructor, contracts allow you to define explicit dependencies for your classes. Some developers prefer to explicitly define their dependencies in this way and therefore prefer to use contracts, while other developers enjoy the convenience of facades. **In general, most applications can use facades without issue during development.**

When to Use Contracts

The decision to use contracts or facades will come down to personal taste and the tastes of your development team. Both contracts and facades can be used to create robust, well-tested Laravel applications. Contracts and facades are not mutually exclusive. Some parts of your applications may use facades while others depend on contracts. As long as you are keeping your class' responsibilities focused, you will notice very few practical differences between using contracts and facades.

In general, most applications can use facades without issue during development. If you are building a package that integrates with multiple PHP frameworks you may wish to use the `illuminate/contracts` package to define your integration with Laravel's services without the need to require Laravel's concrete implementations in your package's `composer.json` file.

How to Use Contracts

So, how do you get an implementation of a contract? It's actually quite simple.

Many types of classes in Laravel are resolved through the [service container](#), including controllers, event listeners, middleware, queued jobs, and even route closures. So, to get an implementation of a contract, you can just "type-hint" the interface in the constructor of the class being resolved.

For example, take a look at this event listener:

```
<?php

namespace App\Listeners;

use App\Events\OrderWasPlaced;
use App\Models\User;
use Illuminate\Contracts\Redis\Factory;

class CacheOrderInformation
{
    /**
     * Create a new event handler instance.
     */
    public function __construct(
        protected Factory $redis,
    ) {}

    /**
     * Handle the event.
     */
    public function handle(OrderWasPlaced $event): void
    {
        // ...
    }
}
```

When the event listener is resolved, the service container will read the type-hints on the constructor of the class, and inject the appropriate value. To learn more about registering things in the service container, check out [its documentation](#).

Contract Reference

This table provides a quick reference to all of the Laravel contracts and their equivalent facades:

Contract	References Facade
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	Gate
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	Auth
Illuminate\Contracts\Auth\Guard	Auth::guard()
Illuminate\Contracts\Auth>PasswordBroker	Password::broker()
Illuminate\Contracts\Auth>PasswordBrokerFactory	Password
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	
Illuminate\Contracts\Auth\UserProvider	
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Bus\QueueingDispatcher	Bus::dispatchToQueue()
Illuminate\Contracts\Broadcasting\Factory	Broadcast
Illuminate\Contracts\Broadcasting\Broadcaster	Broadcast::connection()
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Cache\Factory	Cache
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	Cache::driver()
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	Artisan
Illuminate\Contracts\Container\Container	App
Illuminate\Contracts\Cookie\Factory	Cookie
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Database\ModelIdentifier	
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	Crypt

Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	Storage::cloud()
Illuminate\Contracts\Filesystem\Factory	Storage
Illuminate\Contracts\Filesystem\Filesystem	Storage::disk()
Illuminate\Contracts\Foundation\Application	App
Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Http\Kernel	
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailable	
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Notifications\Dispatcher	Notification
Illuminate\Contracts\Notifications\Factory	Notification
Illuminate\Contracts\Pagination\LengthAwarePaginator	
Illuminate\Contracts\Pagination\Paginator	
Illuminate\Contracts\Pipeline\Hub	
Illuminate\Contracts\Pipeline\Pipeline	Pipeline;
Illuminate\Contracts\Queue\EntityResolver	
Illuminate\Contracts\Queue\Factory	Queue
Illuminate\Contracts\Queue\Job	
Illuminate\Contracts\Queue\Monitor	Queue
Illuminate\Contracts\Queue\Queue	Queue::connection()
Illuminate\Contracts\Queue\QueueableCollection	
Illuminate\Contracts\Queue\QueueableEntity	
Illuminate\Contracts\Queue\ShouldQueue	
Illuminate\Contracts\Redis\Factory	Redis
Illuminate\Contracts\Routing\BindingRegistrar	Route
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Routing\UrlRoutable	
Illuminate\Contracts\Session\Session	Session::driver()
Illuminate\Contracts\Support\Arrayable	

<u>Illuminate\Contracts\Support\Htmlable</u>	
<u>Illuminate\Contracts\Support\Jsonable</u>	
<u>Illuminate\Contracts\Support\MessageBag</u>	
<u>Illuminate\Contracts\Support\MessageProvider</u>	
<u>Illuminate\Contracts\Support\Renderable</u>	
<u>Illuminate\Contracts\Support\Responsable</u>	
<u>Illuminate\Contracts\Translation\Loader</u>	
<u>Illuminate\Contracts\Translation\Translator</u>	Lang
<u>Illuminate\Contracts\Validation\Factory</u>	Validator
<u>Illuminate\Contracts\Validation\ImplicitRule</u>	
<u>Illuminate\Contracts\Validation\Rule</u>	
<u>Illuminate\Contracts\Validation\ValidatesWhenResolved</u>	
<u>Illuminate\Contracts\Validation\Validator</u>	Validator::make()
<u>Illuminate\Contracts\View\Engine</u>	
<u>Illuminate\Contracts\View\Factory</u>	View
<u>Illuminate\Contracts\View\View</u>	View::make()

Events

- [Introduction](#)
- [Registering Events and Listeners](#)
 - [Generating Events and Listeners](#)
 - [Manually Registering Events](#)
 - [Event Discovery](#)
- [Defining Events](#)
- [Defining Listeners](#)
- [Queued Event Listeners](#)
 - [Manually Interacting With the Queue](#)
 - [Queued Event Listeners and Database Transactions](#)
 - [Handling Failed Jobs](#)
- [Dispatching Events](#)
 - [Dispatching Events After Database Transactions](#)
- [Event Subscribers](#)
 - [Writing Event Subscribers](#)
 - [Registering Event Subscribers](#)
- [Testing](#)
 - [Faking a Subset of Events](#)
 - [Scoped Events Fakes](#)

Introduction

Laravel's events provide a simple observer pattern implementation, allowing you to subscribe and listen for various events that occur within your application. Event classes are typically stored in the `app/Events` directory, while their listeners are stored in `app/Listeners`. Don't worry if you don't see these directories in your application as they will be created for you as you generate events and listeners using Artisan console commands.

Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other. For example, you may wish to send a Slack notification to your user each time an order has shipped. Instead of coupling your order processing code to your Slack notification code, you can raise an `App\Events\OrderShipped` event which a listener can receive and use to dispatch a Slack notification.

Registering Events and Listeners

The `App\Providers\EventServiceProvider` included with your Laravel application provides a convenient place to register all of your application's event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). You may add as many events to this array as your application requires. For example, let's add an `OrderShipped` event:

```
use App\Events\OrderShipped;
use App\Listeners\SendShipmentNotification;

/**
 * The event listener mappings for the application.
 *
 * @var array<class-string, array<int, class-string>>
 */
protected $listen = [
    OrderShipped::class => [
        SendShipmentNotification::class,
    ],
];
```



The `event:list` command may be used to display a list of all events and listeners registered by your application.

Generating Events and Listeners

Of course, manually creating the files for each event and listener is cumbersome. Instead, add listeners and events to your `EventServiceProvider` and use the `event:generate` Artisan command. This command will generate any events or listeners that are listed in your `EventServiceProvider` that do not already exist:

```
php artisan event:generate
```

Alternatively, you may use the `make:event` and `make:listener` Artisan commands to generate individual events and listeners:

```
php artisan make:event PodcastProcessed
```

```
php artisan make:listener SendPodcastNotification --event=PodcastProcessed
```

Manually Registering Events

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register class or closure based event listeners manually in the `boot` method of your `EventServiceProvider`:

```

use App\Events\PodcastProcessed;
use App\Listeners\SendPodcastNotification;
use Illuminate\Support\Facades\Event;

/**
 * Register any other events for your application.
 */
public function boot(): void
{
    Event::listen(
        PodcastProcessed::class,
        SendPodcastNotification::class,
    );

    Event::listen(function (PodcastProcessed $event) {
        // ...
    });
}

```

Queueable Anonymous Event Listeners

When registering closure based event listeners manually, you may wrap the listener closure within the `Illuminate\Events\queueable` function to instruct Laravel to execute the listener using the `queue`:

```

use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;

/**
 * Register any other events for your application.
 */
public function boot(): void
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        // ...
    }));
}

```

Like queued jobs, you may use the `onConnection`, `onQueue`, and `delay` methods to customize the execution of the queued listener:

```

Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10)));

```

If you would like to handle anonymous queued listener failures, you may provide a closure to the `catch` method while defining the `queueable` listener. This closure will receive the event instance and the `Throwable` instance that caused the listener's failure:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
use Throwable;

Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->catch(function (PodcastProcessed $event, Throwable $e) {
    // The queued listener failed...
}));
```

Wildcard Event Listeners

You may even register listeners using the `*` as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the event name as their first argument and the entire event data array as their second argument:

```
Event::listen('event.*', function (string $eventName, array $data) {
    // ...
});
```

Event Discovery

Instead of registering events and listeners manually in the `$listen` array of the `EventServiceProvider`, you can enable automatic event discovery. When event discovery is enabled, Laravel will automatically find and register your events and listeners by scanning your application's `Listeners` directory. In addition, any explicitly defined events listed in the `EventServiceProvider` will still be registered.

Laravel finds event listeners by scanning the listener classes using PHP's reflection services. When Laravel finds any listener class method that begins with `handle` or `__invoke`, Laravel will register those methods as event listeners for the event that is type-hinted in the method's signature:

```
use App\Events\PodcastProcessed;

class SendPodcastNotification
{
    /**
     * Handle the given event.
     */
    public function handle(PodcastProcessed $event): void
    {
        // ...
    }
}
```

Event discovery is disabled by default, but you can enable it by overriding the `shouldDiscoverEvents` method of your application's `EventServiceProvider`:

```
/**
 * Determine if events and listeners should be automatically discovered.
 */
public function shouldDiscoverEvents(): bool
{
    return true;
}
```

By default, all listeners within your application's `app/Listeners` directory will be scanned. If you would like to define additional directories to scan, you may override the `discoverEventsWithin` method in your `EventServiceProvider`:

```
/*
 * Get the listener directories that should be used to discover events.
 *
 * @return array<int, string>
 */
protected function discoverEventsWithin(): array
{
    return [
        $this->app->path('Listeners'),
    ];
}
```

Event Discovery In Production

In production, it is not efficient for the framework to scan all of your listeners on every request. Therefore, during your deployment process, you should run the `event:cache` Artisan command to cache a manifest of all of your application's events and listeners. This manifest will be used by the framework to speed up the event registration process. The `event:clear` command may be used to destroy the cache.

Defining Events

An event class is essentially a data container which holds the information related to the event. For example, let's assume an `App\Events\OrderShipped` event receives an [Eloquent ORM](#) object:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     */
    public function __construct(
        public Order $order,
    ) {}
}
```

As you can see, this event class contains no logic. It is a container for the `App\Models\Order` instance that was purchased. The `SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's `serialize` function, such as when utilizing [queued listeners](#).

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive event instances in their `handle` method. The `event:generate` and `make:listener` Artisan commands will automatically import the proper event class and type-hint the event on the `handle` method. Within the `handle` method, you may perform any actions necessary to respond to the event:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * Create the event listener.
     */
    public function __construct()
    {
        // ...
    }

    /**
     * Handle the event.
     */
    public function handle(OrderShipped $event): void
    {
        // Access the order using $event->order...
    }
}
```



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

Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so by returning `false` from your listener's `handle` method.

Queued Event Listeners

Queueing listeners can be beneficial if your listener is going to perform a slow task such as sending an email or making an HTTP request. Before using queued listeners, make sure to [configure your queue](#) and start a queue worker on your server or local development environment.

To specify that a listener should be queued, add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` and `make:listener` Artisan commands already have this interface imported into the current namespace so you can use it immediately:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    // ...
}
```

That's it! Now, when an event handled by this listener is dispatched, the listener will automatically be queued by the event dispatcher using Laravel's [queue system](#). If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has finished processing.

Customizing The Queue Connection, Name, & Delay

If you would like to customize the queue connection, queue name, or queue delay time of an event listener, you may define the `$connection`, `$queue`, or `$delay` properties on your listener class:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * The name of the connection the job should be sent to.
     *
     * @var string|null
     */
    public $connection = 'sq';

    /**
     * The name of the queue the job should be sent to.
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * The time (seconds) before the job should be processed.
     *
     * @var int
     */
    public $delay = 60;
}
```

If you would like to define the listener's queue connection, queue name, or delay at runtime, you may define `viaConnection`, `viaQueue`, or `withDelay` methods on the listener:

```

/**
 * Get the name of the listener's queue connection.
 */
public function viaConnection(): string
{
    return 'sq';
}

/**
 * Get the name of the listener's queue.
 */
public function viaQueue(): string
{
    return 'listeners';
}

/**
 * Get the number of seconds before the job should be processed.
 */
public function withDelay(OrderShipped $event): int
{
    return $event->highPriority ? 0 : 60;
}

```

Conditionally Queueing Listeners

Sometimes, you may need to determine whether a listener should be queued based on some data that are only available at runtime. To accomplish this, a `shouldQueue` method may be added to a listener to determine whether the listener should be queued. If the `shouldQueue` method returns `false`, the listener will not be executed:

```

<?php

namespace App\Listeners;

use App\Events\OrderCreated;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * Reward a gift card to the customer.
     */
    public function handle(OrderCreated $event): void
    {
        // ...
    }

    /**
     * Determine whether the listener should be queued.
     */
    public function shouldQueue(OrderCreated $event): bool
    {
        return $event->order->subtotal >= 5000;
    }
}

```

Manually Interacting With the Queue

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

```
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     */
    public function handle(OrderShipped $event): void
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

Queued Event Listeners and Database Transactions

When queued listeners are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your listener depends on these models, unexpected errors can occur when the job that dispatches the queued listener is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued listener should be dispatched after all open database transactions have been committed by implementing the `ShouldHandleEventsAfterCommit` interface on the listener class:

```
<?php

namespace App\Listeners;

use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue, ShouldHandleEventsAfterCommit
{
    use InteractsWithQueue;
}
```



To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).

Handling Failed Jobs

Sometimes your queued event listeners may fail. If the queued listener exceeds the maximum number of attempts as defined by your queue worker, the `failed` method will be called on your listener. The `failed` method receives the event instance and the `Throwable` that caused the failure:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Throwable;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     */
    public function handle(OrderShipped $event): void
    {
        // ...
    }

    /**
     * Handle a job failure.
     */
    public function failed(OrderShipped $event, Throwable $exception): void
    {
        // ...
    }
}
```

Specifying Queued Listener Maximum Attempts

If one of your queued listeners is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a listener may be attempted.

You may define a `$tries` property on your listener class to specify how many times the listener may be attempted before it is considered to have failed:

```
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * The number of times the queued listener may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

As an alternative to defining how many times a listener may be attempted before it fails, you may define a time at which the listener should no longer be attempted. This allows a listener to be attempted any number of times within a given time frame. To define the time at which a listener should no longer be attempted, add a `retryUntil` method to your listener class. This method should return a `DateTime` instance:

```
use DateTime;

/**
 * Determine the time at which the listener should timeout.
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(5);
}
```

Dispatching Events

To dispatch an event, you may call the static `dispatch` method on the event. This method is made available on the event by the `Illuminate\Foundation\Events\Dispatchable` trait. Any arguments passed to the `dispatch` method will be passed to the event's constructor:

```
<?php

namespace App\Http\Controllers;

use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // Order shipment logic...

        OrderShipped::dispatch($order);

        return redirect('/orders');
    }
}
```

If you would like to conditionally dispatch an event, you may use the `dispatchIf` and `dispatchUnless` methods:

```
OrderShipped::dispatchIf($condition, $order);

OrderShipped::dispatchUnless($condition, $order);
```



When testing, it can be helpful to assert that certain events were dispatched without actually triggering their listeners. Laravel's [built-in testing helpers](#) make it a cinch.

Dispatching Events After Database Transactions

Sometimes, you may want to instruct Laravel to only dispatch an event after the active database transaction has committed. To do so, you may implement the `ShouldDispatchAfterCommit` interface on the event class.

This interface instructs Laravel to not dispatch the event until the current database transaction is committed. If the transaction fails, the event will be discarded. If no database transaction is in progress when the event is dispatched, the event will be dispatched immediately:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Events\ShouldDispatchAfterCommit;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped implements ShouldDispatchAfterCommit
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     */
    public function __construct(
        public Order $order,
    ) {}

}
```

Event Subscribers

Writing Event Subscribers

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

```
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin(Login $event): void {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout(Logout $event): void {}

    /**
     * Register the listeners for the subscriber.
     */
    public function subscribe(Dispatcher $events): void
    {
        $events->listen(
            Login::class,
            [UserEventSubscriber::class, 'handleUserLogin']
        );

        $events->listen(
            Logout::class,
            [UserEventSubscriber::class, 'handleUserLogout']
        );
    }
}
```

If your event listener methods are defined within the subscriber itself, you may find it more convenient to return an array of events and method names from the subscriber's `[subscribe]` method. Laravel will automatically determine the subscriber's class name when registering the event listeners:

```
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin(Login $event): void {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout(Logout $event): void {}

    /**
     * Register the listeners for the subscriber.
     *
     * @return array<string, string>
     */
    public function subscribe(Dispatcher $events): array
    {
        return [
            Login::class => 'handleUserLogin',
            Logout::class => 'handleUserLogout',
        ];
    }
}
```

Registering Event Subscribers

After writing the subscriber, you are ready to register it with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventSubscriber` to the list:

```
<?php

namespace App\Providers;

use App\Listeners\UserEventSubscriber;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

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

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        UserEventSubscriber::class,
    ];
}
```

Testing

When testing code that dispatches events, you may wish to instruct Laravel to not actually execute the event's listeners, since the listener's code can be tested directly and separately of the code that dispatches the corresponding event. Of course, to test the listener itself, you may instantiate a listener instance and invoke the `handle` method directly in your test.

Using the `Event` facade's `fake` method, you may prevent listeners from executing, execute the code under test, and then assert which events were dispatched by your application using the `assertDispatched`, `assertNotDispatched`, and `assertNothingDispatched` methods:

```
<?php

namespace Tests\Feature;

use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function test_orders_can_be_shipped(): void
    {
        Event::fake();

        // Perform order shipping...

        // Assert that an event was dispatched...
        Event::assertDispatched(OrderShipped::class);

        // Assert an event was dispatched twice...
        Event::assertDispatched(OrderShipped::class, 2);

        // Assert an event was not dispatched...
        Event::assertNotDispatched(OrderFailedToShip::class);

        // Assert that no events were dispatched...
        Event::assertNothingDispatched();
    }
}
```

You may pass a closure to the `assertDispatched` or `assertNotDispatched` methods in order to assert that an event was dispatched that passes a given "truth test". If at least one event was dispatched that passes the given truth test then the assertion will be successful:

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

If you would simply like to assert that an event listener is listening to a given event, you may use the `assertListening` method:

```
Event::assertListening(
    OrderShipped::class,
    SendShipmentNotification::class
);
```



After calling `Event::fake()`, no event listeners will be executed. So, if your tests use model factories that rely on events, such as creating a UUID during a model's `creating` event, you should call `Event::fake()` **after** using your factories.

Faking a Subset of Events

If you only want to fake event listeners for a specific set of events, you may pass them to the `fake` or `fakeFor` method:

```
/**
 * Test order process.
 */
public function test_orders_can_be_processed(): void
{
    Event::fake([
        OrderCreated::class,
    ]);

    $order = Order::factory()->create();

    Event::assertDispatched(OrderCreated::class);

    // Other events are dispatched as normal...
    $order->update([...]);
}
```

You may fake all events except for a set of specified events using the `except` method:

```
Event::fake()->except([
    OrderCreated::class,
]);
```

Scoped Event Fakes

If you only want to fake event listeners for a portion of your test, you may use the `fakeFor` method:

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Models\Order;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order process.
     */
    public function test_orders_can_be_processed(): void
    {
        $order = Event::fakeFor(function () {
            $order = Order::factory()->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });

        // Events are dispatched as normal and observers will run ...
        $order->update([...]);
    }
}
```

File Storage

- [Introduction](#)
- [Configuration](#)
 - [The Local Driver](#)
 - [The Public Disk](#)
 - [Driver Prerequisites](#)
 - [Scoped and Read-Only Filesystems](#)
 - [Amazon S3 Compatible Filesystems](#)
- [Obtaining Disk Instances](#)
 - [On-Demand Disks](#)
- [Retrieving Files](#)
 - [Downloading Files](#)
 - [File URLs](#)
 - [Temporary URLs](#)
 - [File Metadata](#)
- [Storing Files](#)
 - [Prepending and Appending To Files](#)
 - [Copying and Moving Files](#)
 - [Automatic Streaming](#)
 - [File Uploads](#)
 - [File Visibility](#)
- [Deleting Files](#)
- [Directories](#)
- [Testing](#)
- [Custom Filesystems](#)

Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful [Flysystem](#) PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3. Even better, it's amazingly simple to switch between these storage options between your local development machine and production server as the API remains the same for each system.

Configuration

Laravel's filesystem configuration file is located at `config/filesystems.php`. Within this file, you may configure all of your filesystem "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file so you can modify the configuration to reflect your storage preferences and credentials.

The `local` driver interacts with files stored locally on the server running the Laravel application while the `s3` driver is used to write to Amazon's S3 cloud storage service.



You may configure as many disks as you like and may even have multiple disks that use the same driver.

The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory defined in your `filesystems` configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would write to `storage/app/example.txt`:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('local')->put('example.txt', 'Contents');
```

The Public Disk

The `public` disk included in your application's `filesystems` configuration file is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores its files in `storage/app/public`.

To make these files accessible from the web, you should create a symbolic link from `public/storage` to `storage/app/public`. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like [Envoyer](#).

To create the symbolic link, you may use the `storage:link` Artisan command:

```
php artisan storage:link
```

Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the `asset` helper:

```
echo asset('storage/file.txt');
```

You may configure additional symbolic links in your `filesystems` configuration file. Each of the configured links will be created when you run the `storage:link` command:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

The `storage:unlink` command may be used to destroy your configured symbolic links:

```
php artisan storage:unlink
```

Driver Prerequisites

S3 Driver Configuration

Before using the S3 driver, you will need to install the Flysystem S3 package via the Composer package manager:

```
composer require league/flysystem-aws-s3-v3 "^3.0" --with-all-dependencies
```

The S3 driver configuration information is located in your `config/filesystems.php` configuration file. This file contains an example configuration array for an S3 driver. You are free to modify this array with your own S3 configuration and credentials. For convenience, these environment variables match the naming convention used by the AWS CLI.

FTP Driver Configuration

Before using the FTP driver, you will need to install the Flysystem FTP package via the Composer package manager:

```
composer require league/flysystem-ftp "^3.0"
```

Laravel's Flysystem integrations work great with FTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an FTP filesystem, you may use the configuration example below:

```
'ftp' => [
    'driver' => 'ftp',
    'host' => env('FTP_HOST'),
    'username' => env('FTP_USERNAME'),
    'password' => env('FTP_PASSWORD'),

    // Optional FTP Settings...
    // 'port' => env('FTP_PORT', 21),
    // 'root' => env('FTP_ROOT'),
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

SFTP Driver Configuration

Before using the SFTP driver, you will need to install the Flysystem SFTP package via the Composer package manager:

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Laravel's Flysystem integrations work great with SFTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an SFTP filesystem, you may use the configuration example below:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => env('SFTP_HOST'),

    // Settings for basic authentication...
    'username' => env('SFTP_USERNAME'),
    'password' => env('SFTP_PASSWORD'),

    // Settings for SSH key based authentication with encryption password...
    'privateKey' => env('SFTP_PRIVATE_KEY'),
    'passphrase' => env('SFTP_PASSPHRASE'),

    // Settings for file / directory permissions...
    'visibility' => 'private', // `private` = 0600, `public` = 0644
    'directory_visibility' => 'private', // `private` = 0700, `public` = 0755

    // Optional SFTP Settings...
    // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
    // 'maxTries' => 4,
    // 'passphrase' => env('SFTP_PASSPHRASE'),
    // 'port' => env('SFTP_PORT', 22),
    // 'root' => env('SFTP_ROOT', ''),
    // 'timeout' => 30,
    // 'useAgent' => true,
],
]
```

Scoped and Read-Only Filesystems

Scoped disks allow you to define a filesystem where all paths are automatically prefixed with a given path prefix. Before creating a scoped filesystem disk, you will need to install an additional Flysystem package via the Composer package manager:

```
composer require league/flysystem-path-prefixing "^3.0"
```

You may create a path scoped instance of any existing filesystem disk by defining a disk that utilizes the `scoped` driver. For example, you may create a disk which scopes your existing `s3` disk to a specific path prefix, and then every file operation using your scoped disk will utilize the specified prefix:

```
's3-videos' => [
    'driver' => 'scoped',
    'disk' => 's3',
    'prefix' => 'path/to/videos',
],
]
```

"Read-only" disks allow you to create filesystem disks that do not allow write operations. Before using the `read-only` configuration option, you will need to install an additional Flysystem package via the Composer package manager:

```
composer require league/flysystem-read-only "^3.0"
```

Next, you may include the `read-only` configuration option in one or more of your disk's configuration arrays:

```
's3-videos' => [
    'driver' => 's3',
    // ...
    'read-only' => true,
],
```

Amazon S3 Compatible Filesystems

By default, your application's `filesystems` configuration file contains a disk configuration for the `s3` disk. In addition to using this disk to interact with Amazon S3, you may use it to interact with any S3 compatible file storage service such as [MinIO](#) or [DigitalOcean Spaces](#).

Typically, after updating the disk's credentials to match the credentials of the service you are planning to use, you only need to update the value of the `endpoint` configuration option. This option's value is typically defined via the `AWS_ENDPOINT` environment variable:

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

MinIO

In order for Laravel's Flysystem integration to generate proper URLs when using MinIO, you should define the `AWS_URL` environment variable so that it matches your application's local URL and includes the bucket name in the URL path:

```
AWS_URL=http://localhost:9000/local
```



Generating temporary storage URLs via the `temporaryUrl` method is not supported when using MinIO.

Obtaining Disk Instances

The `Storage` facade may be used to interact with any of your configured disks. For example, you may use the `put` method on the facade to store an avatar on the default disk. If you call methods on the `Storage` facade without first calling the `disk` method, the method will automatically be passed to the default disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $content);
```

If your application interacts with multiple disks, you may use the `disk` method on the `Storage` facade to work with files on a particular disk:

```
Storage::disk('s3')->put('avatars/1', $content);
```

On-Demand Disks

Sometimes you may wish to create a disk at runtime using a given configuration without that configuration actually being present in your application's `filesystems` configuration file. To accomplish this, you may pass a configuration array to the `Storage` facade's `build` method:

```
use Illuminate\Support\Facades\Storage;

$disk = Storage::build([
    'driver' => 'local',
    'root' => '/path/to/root',
]);

$disk->put('image.jpg', $content);
```

Retrieving Files

The `get` method may be used to retrieve the contents of a file. The raw string contents of the file will be returned by the method. Remember, all file paths should be specified relative to the disk's "root" location:

```
$contents = Storage::get('file.jpg');
```

If the file you are retrieving contains JSON, you may use the `json` method to retrieve the file and decode its contents:

```
$orders = Storage::json('orders.json');
```

The `exists` method may be used to determine if a file exists on the disk:

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

The `missing` method may be used to determine if a file is missing from the disk:

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

Downloading Files

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 Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

File URLs

You may use the `url` method to get the URL for a given file. If you are using the `local` driver, this will typically just prepend `/storage` to the given path and return a relative URL to the file. If you are using the `s3` driver, the fully qualified remote URL will be returned:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::url('file.jpg');
```

When using the `local` driver, all files that should be publicly accessible should be placed in the `storage/app/public` directory. Furthermore, you should [create a symbolic link](#) at `public/storage` which points to the `storage/app/public` directory.



When using the `local` driver, the return value of `url` is not URL encoded. For this reason, we recommend always storing your files using names that will create valid URLs.

URL Host Customization

If you would like to pre-define the host for URLs generated using the `Storage` facade, you may add a `url` option to the disk's configuration array:

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

Temporary URLs

Using the `temporaryUrl` method, you may create temporary URLs to files stored using the `s3` driver. This method accepts a path and a `DateTime` instance specifying when the URL should expire:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

If you need to specify additional [S3 request parameters](#), you may pass the array of request parameters as the third argument to the `temporaryUrl` method:

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
    ]
);
```

If you need to customize how temporary URLs are created for a specific storage disk, you can use the `buildTemporaryUrlsUsing` method. For example, this can be useful if you have a controller that allows you to download files stored via a disk that doesn't typically support temporary URLs. Usually, this method should be called from the `boot` method of a service provider:

```
<?php

namespace App\Providers;

use DateTime;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\URL;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Storage::disk('local')->buildTemporaryUrlsUsing(
            function (string $path, DateTime $expiration, array $options) {
                return URL::temporarySignedRoute(
                    'files.download',
                    $expiration,
                    array_merge($options, ['path' => $path])
                );
            }
        );
    }
}
```

Temporary Upload URLs



The ability to generate temporary upload URLs is only supported by the `s3` driver.

If you need to generate a temporary URL that can be used to upload a file directly from your client-side application, you may use the `temporaryUploadUrl` method. This method accepts a path and a `DateTime` instance specifying when the URL should expire. The `temporaryUploadUrl` method returns an associative array which may be destructured into the upload URL and the headers that should be included with the upload request:

```
use Illuminate\Support\Facades\Storage;

['url' => $url, 'headers' => $headers] = Storage::temporaryUploadUrl(
    'file.jpg', now()->addMinutes(5)
);
```

This method is primarily useful in serverless environments that require the client-side application to directly upload files to a cloud storage system such as Amazon S3.

File Metadata

In addition to reading and writing files, Laravel can also provide information about the files themselves. For example, the `size` method may be used to get the size of a file in bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file.jpg');
```

The MIME type of a given file may be obtained via the `mimeType` method:

```
$mime = Storage::mimeType('file.jpg');
```

File Paths

You may use the `path` method to get the path for a given file. If you are using the `local` driver, this will return the absolute path to the file. If you are using the `s3` driver, this method will return the relative path to the file in the S3 bucket:

```
use Illuminate\Support\Facades\Storage;

$path = Storage::path('file.jpg');
```

Storing Files

The `put` method may be used to store file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Remember, all file paths should be specified relative to the "root" location configured for the disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Failed Writes

If the `put` method (or other "write" operations) is unable to write the file to disk, `false` will be returned:

```
if (! Storage::put('file.jpg', $contents)) {
    // The file could not be written to disk...
}
```

If you wish, you may define the `throw` option within your filesystem disk's configuration array. When this option is defined as `true`, "write" methods such as `put` will throw an instance of `League\Flysystem\UnableToWriteFile` when write operations fail:

```
'public' => [
    'driver' => 'local',
    // ...
    'throw' => true,
],
```

Prepending and Appending To Files

The `prepend` and `append` methods allow you to write to the beginning or end of a file:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Copying and Moving Files

The `copy` method may be used to copy an existing file to a new location on the disk, while the `move` method may be used to rename or move an existing file to a new location:

```
Storage::copy('old/file.jpg', 'new/file.jpg');

Storage::move('old/file.jpg', 'new/file.jpg');
```

Automatic Streaming

Streaming files to storage offers significantly reduced memory usage. If you would like Laravel to automatically manage streaming a given file to your storage location, you may use the `putFile` or `putFileAs` method. This method accepts either an `Illuminate\Http\File` or `Illuminate\Http\UploadedFile` instance and will automatically stream the file to your desired location:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// Automatically generate a unique ID for filename...
$path = Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a filename...
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

There are a few important things to note about the `putFile` method. Note that we only specified a directory name and not a filename. By default, the `putFile` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `putFile` method so you can store the path, including the generated filename, in your database.

The `putFile` and `putFileAs` methods also accept an argument to specify the "visibility" of the stored file. This is particularly useful if you are storing the file on a cloud disk such as Amazon S3 and would like the file to be publicly accessible via generated URLs:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as photos and documents. Laravel makes it very easy to store uploaded files using the `store` method on an uploaded file instance. Call the `store` method with the path at which you wish to store the uploaded file:

```
<?php

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     */
    public function update(Request $request): string
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

There are a few important things to note about this example. Note that we only specified a directory name, not a filename. By default, the `store` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `store` method so you can store the path, including the generated filename, in your database.

You may also call the `putFile` method on the `Storage` facade to perform the same file storage operation as the example above:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying a File Name

If you do not want a filename to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the filename, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

You may also use the `putFileAs` method on the `Storage` facade, which will perform the same file storage operation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```



Unprintable and invalid unicode characters will automatically be removed from file paths. Therefore, you may wish to sanitize your file paths before passing them to Laravel's file storage methods. File paths are normalized using the `League\Flysystem\WhitespacePathNormalizer::normalizePath` method.

Specifying a Disk

By default, this uploaded file's `store` method will use your default disk. If you would like to specify another disk, pass the disk name as the second argument to the `store` method:

```
$path = $request->file('avatar')->store(
    'avatars/' . $request->user()->id, 's3'
);
```

If you are using the `storeAs` method, you may pass the disk name as the third argument to the method:

```
$path = $request->file('avatar')->storeAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Other Uploaded File Information

If you would like to get the original name and extension of the uploaded file, you may do so using the `getClientOriginalName` and `getClientOriginalExtension` methods:

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

$name = $file->getClientOriginalName();
$extension = $file->getClientOriginalExtension();
```

However, keep in mind that the `getClientOriginalName` and `getClientOriginalExtension` methods are considered unsafe, as the file name and extension may be tampered with by a malicious user. For this reason, you should typically prefer the `hashName` and `extension` methods to get a name and an extension for the given file upload:

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

$name = $file->hashName(); // Generate a unique, random name...
$extension = $file->extension(); // Determine the file's extension based on the file's MIME type...
```

File Visibility

In Laravel's Flysystem integration, "visibility" is an abstraction of file permissions across multiple platforms. Files may either be declared `public` or `private`. When a file is declared `public`, you are indicating that the file should generally be accessible to others. For example, when using the S3 driver, you may retrieve URLs for `public` files.

You can set the visibility when writing the file via the `put` method:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

If the file has already been stored, its visibility can be retrieved and set via the `getVisibility` and `setVisibility` methods:

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

When interacting with uploaded files, you may use the `storePublicly` and `storePubliclyAs` methods to store the uploaded file with `public` visibility:

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Local Files and Visibility

When using the `local` driver, `public` `visibility` translates to `0755` permissions for directories and `0644` permissions for files. You can modify the permissions mappings in your application's `filesystems` configuration file:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0644,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0755,
            'private' => 0700,
        ],
    ],
],
```

Deleting Files

The `delete` method accepts a single filename or an array of files to delete:

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file.jpg', 'file2.jpg']);
```

If necessary, you may specify the disk that the file should be deleted from:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('path/file.jpg');
```

Directories

Get All Files Within a Directory

The `files` method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all subdirectories, you may use the `allFiles` method:

```
use Illuminate\Support\Facades\Storage;  
  
$files = Storage::files($directory);  
  
$files = Storage::allFiles($directory);
```

Get All Directories Within a Directory

The `directories` method returns an array of all the directories within a given directory. Additionally, you may use the `allDirectories` method to get a list of all directories within a given directory and all of its subdirectories:

```
$directories = Storage::directories($directory);  
  
$directories = Storage::allDirectories($directory);
```

Create a Directory

The `makeDirectory` method will create the given directory, including any needed subdirectories:

```
Storage::makeDirectory($directory);
```

Delete a Directory

Finally, the `deleteDirectory` method may be used to remove a directory and all of its files:

```
Storage::deleteDirectory($directory);
```

Testing

The `Storage` facade's `fake` method allows you to easily generate a fake disk that, combined with the file generation utilities of the `Illuminate\Http\UploadedFile` class, greatly simplifies the testing of file uploads. For example:

```
<?php

namespace Tests\Feature;

use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_albums_can_be_uploaded(): void
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);

        // Assert one or more files were stored...
        Storage::disk('photos')->assertExists('photo1.jpg');
        Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

        // Assert one or more files were not stored...
        Storage::disk('photos')->assertMissing('missing.jpg');
        Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);

        // Assert that a given directory is empty...
        Storage::disk('photos')->assertDirectoryEmpty('/wallpapers');
    }
}
```

By default, the `fake` method will delete all files in its temporary directory. If you would like to keep these files, you may use the "persistentFake" method instead. For more information on testing file uploads, you may consult the [HTTP testing documentation's information on file uploads](#).



The `image` method requires the [GD extension](#).

Custom Filesystems

Laravel's Flysystem integration provides support for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application.

In order to define a custom filesystem you will need a Flysystem adapter. Let's add a community maintained Dropbox adapter to our project:

```
composer require spatie/flysystem-dropbox
```

Next, you can register the driver within the `boot` method of one of your application's [service providers](#). To accomplish this, you should use the `extend` method of the `Storage` facade:

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Foundation\Application;
use Illuminate\Filesystem\FilesystemAdapter;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Storage::extend('dropbox', function (Application $app, array $config) {
            $adapter = new DropboxAdapter(new DropboxClient(
                $config['authorization_token']
            ));

            return new FilesystemAdapter(
                new Filesystem($adapter, $config),
                $adapter,
                $config
            );
        });
    }
}
```

The first argument of the `extend` method is the name of the driver and the second is a closure that receives the `$app` and `$config` variables. The closure must return an instance of `Illuminate\Filesystem\FilesystemAdapter`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Once you have created and registered the extension's service provider, you may use the `dropbox` driver in your `config/filesystems.php` configuration file.

Helpers

- [Introduction](#)
- [Available Methods](#)
- [Other Utilities](#)
 - [Benchmarking](#)
 - [Dates](#)
 - [Lottery](#)
 - [Pipeline](#)
 - [Sleep](#)

Introduction

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

Available Methods

Arrays & Objects

Arr::accessible	Arr::join	Arr::sortRecursive
Arr::add	Arr::keyBy	Arr::sortRecursiveDesc
Arr::collapse	Arr::last	Arr::take
Arr::crossJoin	Arr::map	Arr::toCssClasses
Arr::divide	Arr::mapWithKeys	Arr::toCssStyles
Arr::dot	Arr::only	Arr::undot
Arr::except	Arr::pluck	Arr::where
Arr::exists	Arr::prepend	Arr::whereNotNull
Arr::first	Arr::prependKeysWith	Arr::wrap
Arr::flatten	Arr::pull	data_fill
Arr::forget	Arr::query	data_get
Arr::get	Arr::random	data_set
Arr::has	Arr::set	data_forget
Arr::hasAny	Arr::shuffle	head
Arr::isAssoc	Arr::sort	last
Arr::isList	Arr::sortDesc	

Numbers

Number::abbreviate	Number::forHumans	Number::spell
Number::clamp	Number::format	Number::useLocale
Number::currency	Number::ordinal	Number::withLocale
Number::fileSize	Number::percentage	

Paths

app_path	database_path	public_path
base_path	lang_path	resource_path
config_path	mix	storage_path

URLs

action	secure_asset	url
asset	secure_url	
route	to_route	

Miscellaneous

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

Arrays & Objects

Arr::accessible()

The `Arr::accessible` method determines if the given value is array accessible:

```
use Illuminate\Support\Arr;
use Illuminate\Support\Collection;

$isAccessible = Arr::accessible(['a' => 1, 'b' => 2]);
// true

$isAccessible = Arr::accessible(new Collection);
// true

$isAccessible = Arr::accessible('abc');
// false

$isAccessible = Arr::accessible(new stdClass);
// false
```

Arr::add()

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

```
use Illuminate\Support\Arr;

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

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

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

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

Arr::collapse()

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

```
use Illuminate\Support\Arr;

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

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

Arr::crossJoin()

The `Arr::crossJoin` method cross joins the given arrays, returning a Cartesian product with all possible permutations:

```
use Illuminate\Support\Arr;

$matrix = Arr::crossJoin([1, 2], ['a', 'b']);

/*
[
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
]
*/

$matrix = Arr::crossJoin([1, 2], ['a', 'b'], ['I', 'II']);

/*
[
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
    [2, 'b', 'I'],
    [2, 'b', 'II'],
]
*/
```

Arr::divide()

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

```
use Illuminate\Support\Arr;

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

// $keys: ['name']

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

`Arr::dot()`

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

```
use Illuminate\Support\Arr;

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

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

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

`Arr::except()`

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

```
use Illuminate\Support\Arr;

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

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

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

`Arr::exists()`

The `Arr::exists` method checks that the given key exists in the provided array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'John Doe', 'age' => 17];

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

// true

$exists = Arr::exists($array, 'salary');

// false
```

`Arr::first()`

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

```
use Illuminate\Support\Arr;

$array = [100, 200, 300];

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

// 200
```

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

```
use Illuminate\Support\Arr;

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

`Arr::flatten()`

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

```
use Illuminate\Support\Arr;

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

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

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

`Arr::forget()`

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

```
use Illuminate\Support\Arr;

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

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

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

`Arr::get()`

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

```
use Illuminate\Support\Arr;

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

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

// 100
```

The `Arr::get` method also accepts a default value, which will be returned if the specified key is not present in the array:

```
use Illuminate\Support\Arr;

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

// 0
```

Arr::has()

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

```
use Illuminate\Support\Arr;

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

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

// true

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

// false
```

Arr::hasAny()

The `Arr::hasAny` method checks whether any item in a given set exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

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

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

// true

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

// true

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

// false
```

Arr::isAssoc()

The `Arr::isAssoc` method returns `true` if the given array is an associative array. An array is considered "associative" if it doesn't have sequential numerical keys beginning with zero:

```
use Illuminate\Support\Arr;

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

// true

$isAssoc = Arr::isAssoc([1, 2, 3]);

// false
```

Arr::isList()

The `Arr::isList` method returns `true` if the given array's keys are sequential integers beginning from zero:

```
use Illuminate\Support\Arr;

$isList = Arr::isList(['foo', 'bar', 'baz']);

// true

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

// false
```

Arr::join()

The `Arr::join` method joins array elements with a string. Using this method's second argument, you may also specify the joining string for the final element of the array:

```
use Illuminate\Support\Arr;

$array = ['Tailwind', 'Alpine', 'Laravel', 'Livewire'];

$joined = Arr::join($array, ', ');

// Tailwind, Alpine, Laravel, Livewire

$joined = Arr::join($array, ', ', ' and ');

// Tailwind, Alpine, Laravel and Livewire
```

Arr::keyBy()

The `Arr::keyBy` method keys the array by the given key. If multiple items have the same key, only the last one will appear in the new array:

```
use Illuminate\Support\Arr;

$array = [
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
];

$keyed = Arr::keyBy($array, 'product_id');

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

Arr::last()

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

```
use Illuminate\Support\Arr;

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

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

// 300
```

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

```
use Illuminate\Support\Arr;

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

Arr::map()

The `Arr::map` method iterates through the array and passes each value and key to the given callback. The array value is replaced by the value returned by the callback:

```
use Illuminate\Support\Arr;

$array = ['first' => 'james', 'last' => 'kirk'];

$mapped = Arr::map($array, function (string $value, string $key) {
    return ucfirst($value);
});

// ['first' => 'James', 'last' => 'Kirk']
```

Arr::mapWithKeys()

The `Arr::mapWithKeys` method iterates through the array and passes each value to the given callback. The callback should return an associative array containing a single key / value pair:

```
use Illuminate\Support\Arr;

$array = [
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
];

$mapped = Arr::mapWithKeys($array, function (array $item, int $key) {
    return [$item['email'] => $item['name']];
});

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/
```

Arr::only()

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

```
use Illuminate\Support\Arr;

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

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

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

Arr::pluck()

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

```
use Illuminate\Support\Arr;

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

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

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

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

```
use Illuminate\Support\Arr;

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

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

Arr::prepend()

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

```
use Illuminate\Support\Arr;

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

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

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

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

```
use Illuminate\Support\Arr;

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

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

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

Arr::prependKeysWith()

The `Arr::prependKeysWith` prepends all key names of an associative array with the given prefix:

```
use Illuminate\Support\Arr;

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

$keyed = Arr::prependKeysWith($array, 'product.');

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

Arr::pull()

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

```
use Illuminate\Support\Arr;

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

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

// $name: Desk

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

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

```
use Illuminate\Support\Arr;

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

Arr::query()

The `Arr::query` method converts the array into a query string:

```
use Illuminate\Support\Arr;

$array = [
    'name' => 'Taylor',
    'order' => [
        'column' => 'created_at',
        'direction' => 'desc'
    ]
];

Arr::query($array);

// name=Taylor&order[column]=created_at&order[direction]=desc
```

Arr::random()

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

```
use Illuminate\Support\Arr;

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

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

// 4 - (retrieved randomly)
```

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

```
use Illuminate\Support\Arr;

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

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

Arr::set()

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

```
use Illuminate\Support\Arr;

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

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

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

Arr::shuffle()

The `Arr::shuffle` method randomly shuffles the items in the array:

```
use Illuminate\Support\Arr;

$array = Arr::shuffle([1, 2, 3, 4, 5]);

// [3, 2, 5, 1, 4] - (generated randomly)
```

Arr::sort()

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

```
use Illuminate\Support\Arr;

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

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

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

You may also sort the array by the results of a given closure:

```
use Illuminate\Support\Arr;

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

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

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

Arr::sortDesc()

The `Arr::sortDesc` method sorts an array in descending order by its values:

```
use Illuminate\Support\Arr;

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

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

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

You may also sort the array by the results of a given closure:

```
use Illuminate\Support\Arr;

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

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

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

Arr::sortRecursive()

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

```
use Illuminate\Support\Arr;

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

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

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

If you would like the results sorted in descending order, you may use the `Arr::sortRecursiveDesc` method.

```
$sorted = Arr::sortRecursiveDesc($array);
```

Arr::take()

The `Arr::take` method returns a new array with the specified number of items:

```
use Illuminate\Support\Arr;

$array = [0, 1, 2, 3, 4, 5];

$chunk = Arr::take($array, 3);

// [0, 1, 2]
```

You may also pass a negative integer to take the specified number of items from the end of the array:

```
$array = [0, 1, 2, 3, 4, 5];

$chunk = Arr::take($array, -2);

// [4, 5]
```

Arr::toCssClasses()

The `Arr::toCssClasses` method conditionally compiles a CSS class string. The method 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:

```
use Illuminate\Support\Arr;

$isActive = false;
$hasError = true;

$array = ['p-4', 'font-bold' => $isActive, 'bg-red' => $hasError];

$classes = Arr::toCssClasses($array);

/*
  'p-4 bg-red'
*/
```

Arr::toCssStyles()

The `Arr::toCssStyles` conditionally compiles a CSS style string. The method 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:

```
use Illuminate\Support\Arr;

$hasColor = true;

$array = ['background-color: blue', 'color: blue' => $hasColor];

$classes = Arr::toCssStyles($array);

/*
  'background-color: blue; color: blue;'
*/
```

This method powers Laravel's functionality allowing [merging classes with a Blade component's attribute bag](#) as well as the `@class` [Blade directive](#).

Arr::undot()

The `Arr::undot` method expands a single-dimensional array that uses "dot" notation into a multi-dimensional array:

```
use Illuminate\Support\Arr;

$array = [
    'user.name' => 'Kevin Malone',
    'user.occupation' => 'Accountant',
];

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

// ['user' => ['name' => 'Kevin Malone', 'occupation' => 'Accountant']]
```

Arr::where()

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

```
use Illuminate\Support\Arr;

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

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

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

Arr::whereNotNull()

The `Arr::whereNotNull` method removes all `null` values from the given array:

```
use Illuminate\Support\Arr;

$array = [0, null];

$filtered = Arr::whereNotNull($array);

// [0 => 0]
```

Arr::wrap()

The `Arr::wrap` method wraps the given value in an array. If the given value is already an array it will be returned without modification:

```
use Illuminate\Support\Arr;

$string = 'Laravel';

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

// ['Laravel']
```

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

```
use Illuminate\Support\Arr;

$array = Arr::wrap(null);

// []
```

data_fill()

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

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

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

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

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

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

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

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

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

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

`data_get()`

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

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

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

// 100
```

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

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

// 0
```

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

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

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

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

data_set()

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

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

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

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

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

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

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

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

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

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

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

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

data_forget()

The `data_forget` function removes a value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_forget($data, 'products.desk.price');  
  
// ['products' => ['desk' => []]]
```

This function also accepts wildcards using asterisks and will remove values on the target accordingly:

```
$data = [  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 100],  
        ['name' => 'Desk 2', 'price' => 150],  
    ],  
];  
  
data_forget($data, 'products.*.price');  
  
/*  
 [  
     'products' => [  
         ['name' => 'Desk 1'],  
         ['name' => 'Desk 2'],  
     ],  
 ]  
*/
```

head()

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

```
$array = [100, 200, 300];  
  
$first = head($array);  
  
// 100
```

last()

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

```
$array = [100, 200, 300];  
  
$last = last($array);  
  
// 300
```

Numbers

Number::abbreviate()

The `Number::abbreviate` method returns the human-readable format of the provided numerical value, with an abbreviation for the units:

```
use Illuminate\Support\Number;

$number = Number::abbreviate(1000);

// 1K

$number = Number::abbreviate(489939);

// 490K

$number = Number::abbreviate(1230000, precision: 2);

// 1.23M
```

Number::clamp()

The `Number::clamp` method ensures a given number stays within a specified range. If the number is lower than the minimum, the minimum value is returned. If the number is higher than the maximum, the maximum value is returned:

```
use Illuminate\Support\Number;

$number = Number::clamp(105, min: 10, max: 100);

// 100

$number = Number::clamp(5, min: 10, max: 100);

// 10

$number = Number::clamp(10, min: 10, max: 100);

// 10

$number = Number::clamp(20, min: 10, max: 100);

// 20
```

Number::currency()

The `Number::currency` method returns the currency representation of the given value as a string:

```
use Illuminate\Support\Number;

$currency = Number::currency(1000);

// $1,000

$currency = Number::currency(1000, in: 'EUR');

// €1,000

$currency = Number::currency(1000, in: 'EUR', locale: 'de');

// 1.000 €
```

Number::fileSize()

The `Number::fileSize` method returns the file size representation of the given byte value as a string:

```
use Illuminate\Support\Number;

$size = Number::fileSize(1024);

// 1 KB

$size = Number::fileSize(1024 * 1024);

// 1 MB

$size = Number::fileSize(1024, precision: 2);

// 1.00 KB
```

Number::forHumans()

The `Number::forHumans` method returns the human-readable format of the provided numerical value:

```
use Illuminate\Support\Number;

$number = Number::forHumans(1000);

// 1 thousand

$number = Number::forHumans(489939);

// 490 thousand

$number = Number::forHumans(1230000, precision: 2);

// 1.23 million
```

Number::format()

The `Number::format` method formats the given number into a locale specific string:

```
use Illuminate\Support\Number;

$number = Number::format(100000);

// 100,000

$number = Number::format(100000, precision: 2);

// 100,000.00

$number = Number::format(100000.123, maxPrecision: 2);

// 100,000.12

$number = Number::format(100000, locale: 'de');

// 100.000
```

Number::ordinal()

The `Number::ordinal` method returns a number's ordinal representation:

```
use Illuminate\Support\Number;

$number = Number::ordinal(1);

// 1st

$number = Number::ordinal(2);

// 2nd

$number = Number::ordinal(21);

// 21st
```

Number::percentage()

The `Number::percentage` method returns the percentage representation of the given value as a string:

```
use Illuminate\Support\Number;

$percentage = Number::percentage(10);

// 10%

$percentage = Number::percentage(10, precision: 2);

// 10.00%

$percentage = Number::percentage(10.123, maxPrecision: 2);

// 10.12%

$percentage = Number::percentage(10, precision: 2, locale: 'de');

// 10,00%
```

Number::spell()

The `Number::spell` method transforms the given number into a string of words:

```
use Illuminate\Support\Number;

$number = Number::spell(102);

// one hundred and two

$number = Number::spell(88, locale: 'fr');

// quatre-vingt-huit
```

The `after` argument allows you to specify a value after which all numbers should be spelled out:

```
$number = Number::spell(10, after: 10);

// 10

$number = Number::spell(11, after: 10);

// eleven
```

The `until` argument allows you to specify a value before which all numbers should be spelled out:

```
$number = Number::spell(5, until: 10);

// five

$number = Number::spell(10, until: 10);

// 10
```

Number::useLocale()

The `Number::useLocale` method sets the default number locale globally, which affects how numbers and currency are formatted by subsequent invocations to the `Number` class's methods:

```
use Illuminate\Support\Number;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Number::useLocale('de');
}
```

`Number::withLocale()`

The `Number::withLocale` method executes the given closure using the specified locale and then restores the original locale after the callback has executed:

```
use Illuminate\Support\Number;

$number = Number::withLocale('de', function () {
    return Number::format(1500);
});
```

Paths

`app_path()`

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

```
$path = app_path();

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

`base_path()`

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

```
$path = base_path();

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

`config_path()`

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

```
$path = config_path();

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

`database_path()`

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

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

`lang_path()`

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

```
$path = lang_path();
$path = lang_path('en/messages.php');
```



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

`mix()`

The `mix` function returns the path to a [versioned Mix file](#):

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

`public_path()`

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

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

`resource_path()`

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

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

`storage_path()`

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

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

URLs

`action()`

The `action` function generates a URL for the given controller action:

```
use App\Http\Controllers\HomeController;
$url = action([HomeController::class, 'index']);
```

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

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

`asset()`

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

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

You can configure the asset URL host by setting the `ASSET_URL` variable in your `.env` file. This can be useful if you host your assets on an external service like Amazon S3 or another CDN:

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

`route()`

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

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

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

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

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

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

`secure_asset()`

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

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

secure_url()

The `secure_url` function generates a fully qualified HTTPS URL to the given path. Additional URL segments may be passed in the function's second argument:

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

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

to_route()

The `to_route` function generates a redirect HTTP response for a given named route:

```
return to_route('users.show', ['user' => 1]);
```

If necessary, you may pass the HTTP status code that should be assigned to the redirect and any additional response headers as the third and fourth arguments to the `to_route` method:

```
return to_route('users.show', ['user' => 1], 302, ['X-Framework' => 'Laravel']);
```

url()

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

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

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

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

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

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

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

Miscellaneous

abort()

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

```
abort(403);
```

You may also provide the exception's message and custom HTTP response headers that should be sent to the browser:

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

abort_if()

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

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

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

`abortUnless()`

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

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

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

`app()`

The `app` function returns the [service container](#) instance:

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

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

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

`auth()`

The `auth` function returns an [authenticator](#) instance. You may use it as an alternative to the `Auth` facade:

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

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

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

`back()`

The `back` function generates a [redirect HTTP response](#) to the user's previous location:

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

return back();
```

`bcrypt()`

The `bcrypt` function [hashes](#) the given value using Bcrypt. You may use this function as an alternative to the `Hash` facade:

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

`blank()`

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

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

// true

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

// false
```

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

`broadcast()`

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

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

broadcast(new UserRegistered($user))->toOthers();
```

`cache()`

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

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

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

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

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

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

`class_uses_recursive()`

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

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

`collect()`

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

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

`config()`

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

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

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

You may set configuration variables at runtime by passing an array of key / value pairs. However, note that this function only affects the configuration value for the current request and does not update your actual configuration values:

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

cookie()

The `cookie` function creates a new [cookie](#) instance:

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

csrf_field()

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

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

csrf_token()

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

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

decrypt()

The `decrypt` function [decrypts](#) the given value. You may use this function as an alternative to the `Crypt` facade:

```
$password = decrypt($value);
```

dd()

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

```
dd($value);

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

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

dispatch()

The `dispatch` function pushes the given `job` onto the Laravel [job queue](#):

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

dispatch_sync()

The `dispatch_sync` function pushes the given job to the `sync` queue so that it is processed immediately:

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

dump()

The `dump` function dumps the given variables:

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

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

encrypt()

The `encrypt` function encrypts the given value. You may use this function as an alternative to the `Crypt` facade:

```
$secret = encrypt('my-secret-value');
```

env()

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

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



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

event()

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

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

fake()

The `fake` function resolves a Faker singleton from the container, which can be useful when creating fake data in model factories, database seeding, tests, and prototyping views:

```
@for($i = 0; $i < 10; $i++)
<dl>
    <dt>Name</dt>
    <dd>{{ fake()>name() }}</dd>

    <dt>Email</dt>
    <dd>{{ fake()>unique()>safeEmail() }}</dd>
</dl>
@endfor
```

By default, the `fake` function will utilize the `app.faker.locale` configuration option in your `config/app.php` configuration file; however, you may also specify the locale by passing it to the `fake` function. Each locale will resolve an individual singleton:

```
fake('nl_NL')>name()
```

`filled()`

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

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

// true

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

// false
```

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

`info()`

The `info` function will write information to your application's `log`:

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

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

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

`logger()`

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

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

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

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

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

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

`method_field()`

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

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

`now()`

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

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

`old()`

The `old` function [retrieves](#) an old input value flashed into the session:

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

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

Since the "default value" provided as the second argument to the `old` function is often an attribute of an Eloquent model, Laravel allows you to simply pass the entire Eloquent model as the second argument to the `old` function. When doing so, Laravel will assume the first argument provided to the `old` function is the name of the Eloquent attribute that should be considered the "default value":

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

// Is equivalent to...

{{ old('name', $user) }}
```

`optional()`

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

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

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

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

```
return optional(User::find($id), function (User $user) {
    return $user->name;
});
```

policy()

The `policy` method retrieves a [policy](#) instance for a given class:

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

redirect()

The `redirect` function returns a [redirect HTTP response](#), or returns the redirector instance if called with no arguments:

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

return redirect('/home');

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

report()

The `report` function will report an exception using your [exception handler](#):

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

The `report` function also accepts a string as an argument. When a string is given to the function, the function will create an exception with the given string as its message:

```
report('Something went wrong.');
```

report_if()

The `report_if` function will report an exception using your [exception handler](#) if the given condition is `true`:

```
report_if($shouldReport, $e);

report_if($shouldReport, 'Something went wrong.');
```

report_unless()

The `report_unless` function will report an exception using your [exception handler](#) if the given condition is `false`:

```
report_unless($reportingDisabled, $e);

report_unless($reportingDisabled, 'Something went wrong.');
```

request()

The `request` function returns the current [request](#) instance or obtains an input field's value from the current request:

```
$request = request();

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

rescue()

The `rescue` function executes the given closure and catches any exceptions that occur during its execution. All exceptions that are caught will be sent to your [exception handler](#); however, the request will continue processing:

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

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

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

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

A `report` argument may be provided to the `rescue` function to determine if the exception should be reported via the `report` function:

```
return rescue(function () {
    return $this->method();
}, report: function (Throwable $throwable) {
    return $throwable instanceof InvalidArgumentException;
});
```

resolve()

The `resolve` function resolves a given class or interface name to an instance using the [service container](#):

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

response()

The `response` function creates a [response](#) instance or obtains an instance of the response factory:

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

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

retry()

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

automatically be retried. If the maximum attempt count is exceeded, the exception will be thrown:

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

If you would like to manually calculate the number of milliseconds to sleep between attempts, you may pass a closure as the third argument to the `retry` function:

```
use Exception;

return retry(5, function () {
    // ...
}, function (int $attempt, Exception $exception) {
    return $attempt * 100;
});
```

For convenience, you may provide an array as the first argument to the `retry` function. This array will be used to determine how many milliseconds to sleep between subsequent attempts:

```
return retry([100, 200], function () {
    // Sleep for 100ms on first retry, 200ms on second retry...
});
```

To only retry under specific conditions, you may pass a closure as the fourth argument to the `retry` function:

```
use Exception;

return retry(5, function () {
    // ...
}, 100, function (Exception $exception) {
    return $exception instanceof RetryException;
});
```

session()

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

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

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

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

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

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

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

tap()

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

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

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

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

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

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

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

`throw_if()`

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

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

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

`throw_unless()`

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

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

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

`today()`

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

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

trait_uses_recursive()

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

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

transform()

The `transform` function executes a closure on a given value if the value is not `blank` and then returns the return value of the closure:

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

$result = transform(5, $callback);

// 10
```

A default value or closure may be passed as the third argument to the function. This value will be returned if the given value is blank:

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

// The value is blank
```

validator()

The `validator` function creates a new `Validator` instance with the given arguments. You may use it as an alternative to the `Validator` facade:

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

value()

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

```
$result = value(true);

// true

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

// false
```

Additional arguments may be passed to the `value` function. If the first argument is a closure then the additional parameters will be passed to the closure as arguments, otherwise they will be ignored:

```
$result = value(function (string $name) {
    return $name;
}, 'Taylor');

// 'Taylor'
```

view()

The `view` function retrieves a `view` instance:

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

with()

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

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

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

Other Utilities

Benchmarking

Sometimes you may wish to quickly test the performance of certain parts of your application. On those occasions, you may utilize the `Benchmark` support class to measure the number of milliseconds it takes for the given callbacks to complete:

```
<?php

use App\Models\User;
use Illuminate\Support\Benchmark;

Benchmark::dd(fn () => User::find(1)); // 0.1 ms

Benchmark::dd([
    'Scenario 1' => fn () => User::count(), // 0.5 ms
    'Scenario 2' => fn () => User::all()->count(), // 20.0 ms
]);
```

By default, the given callbacks will be executed once (one iteration), and their duration will be displayed in the browser / console.

To invoke a callback more than once, you may specify the number of iterations that the callback should be invoked as the second argument to the method. When executing a callback more than once, the `Benchmark` class will return the average amount of milliseconds it took to execute the callback across all iterations:

```
Benchmark::dd(fn () => User::count(), iterations: 10); // 0.5 ms
```

Sometimes, you may want to benchmark the execution of a callback while still obtaining the value returned by the callback. The `value` method will return a tuple containing the value returned by the callback and the amount of milliseconds it took to execute the callback:

```
[$count, $duration] = Benchmark::value(fn () => User::count());
```

Dates

Laravel includes `Carbon`, a powerful date and time manipulation library. To create a new `Carbon` instance, you may invoke the `now` function. This function is globally available within your Laravel application:

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

Or, you may create a new `Carbon` instance using the `Illuminate\Support\Carbon` class:

```
use Illuminate\Support\Carbon;

$now = Carbon::now();
```

For a thorough discussion of Carbon and its features, please consult the [official Carbon documentation](#).

Lottery

Laravel's lottery class may be used to execute callbacks based on a set of given odds. This can be particularly useful when you only want to execute code for a percentage of your incoming requests:

```
use Illuminate\Support\Lottery;

Lottery::odds(1, 20)
    ->winner(fn () => $user->won())
    ->loser(fn () => $user->lost())
    ->choose();
```

You may combine Laravel's lottery class with other Laravel features. For example, you may wish to only report a small percentage of slow queries to your exception handler. And, since the lottery class is callable, we may pass an instance of the class into any method that accepts callables:

```
use Carbon\CarbonInterval;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Lottery;

DB::whenQueryingForLongerThan(
    CarbonInterval::seconds(2),
    Lottery::odds(1, 100)->winner(fn () => report('Querying > 2 seconds.')),
);
```

Testing Lotteries

Laravel provides some simple methods to allow you to easily test your application's lottery invocations:

```
// Lottery will always win...
Lottery::alwaysWin();

// Lottery will always lose...
Lottery::alwaysLose();

// Lottery will win then lose, and finally return to normal behavior...
Lottery::fix([true, false]);

// Lottery will return to normal behavior...
Lottery::determineResultsNormally();
```

Pipeline

Laravel's `Pipeline` facade provides a convenient way to "pipe" a given input through a series of invokable classes, closures, or callables, giving each class the opportunity to inspect or modify the input and invoke the next callable in the pipeline:

```
use Closure;
use App\Models\User;
use Illuminate\Support\Facades\Pipeline;

$user = Pipeline::send($user)
    ->through([
        function (User $user, Closure $next) {
            // ...

            return $next($user);
        },
        function (User $user, Closure $next) {
            // ...

            return $next($user);
        },
    ])
->then(fn (User $user) => $user);
```

As you can see, each invokable class or closure in the pipeline is provided the input and a `$next` closure. Invoking the `$next` closure will invoke the next callable in the pipeline. As you may have noticed, this is very similar to [middleware](#).

When the last callable in the pipeline invokes the `$next` closure, the callable provided to the `then` method will be invoked. Typically, this callable will simply return the given input.

Of course, as discussed previously, you are not limited to providing closures to your pipeline. You may also provide invokable classes. If a class name is provided, the class will be instantiated via Laravel's [service container](#), allowing dependencies to be injected into the invokable class:

```
$user = Pipeline::send($user)
    ->through([
        GenerateProfilePhoto::class,
        ActivateSubscription::class,
        SendWelcomeEmail::class,
    ])
->then(fn (User $user) => $user);
```

Sleep

Laravel's `Sleep` class is a light-weight wrapper around PHP's native `sleep` and `usleep` functions, offering greater testability while also exposing a developer friendly API for working with time:

```
use Illuminate\Support\Sleep;

$waiting = true;

while ($waiting) {
    Sleep::for(1)->second();

    $waiting = /* ... */;
}
```

The `Sleep` class offers a variety of methods that allow you to work with different units of time:

```
// Pause execution for 90 seconds...
Sleep::for(1.5)->minutes();

// Pause execution for 2 seconds...
Sleep::for(2)->seconds();

// Pause execution for 500 milliseconds...
Sleep::for(500)->milliseconds();

// Pause execution for 5,000 microseconds...
Sleep::for(5000)->microseconds();

// Pause execution until a given time...
Sleep::until(now()->addMinute());

// Alias of PHP's native "sleep" function...
Sleep::sleep(2);

// Alias of PHP's native "usleep" function...
Sleep::usleep(5000);
```

To easily combine units of time, you may use the `and` method:

```
Sleep::for(1)->second()->and(10)->milliseconds();
```

Testing Sleep

When testing code that utilizes the `Sleep` class or PHP's native sleep functions, your test will pause execution. As you might expect, this makes your test suite significantly slower. For example, imagine you are testing the following code:

```
$waiting = /* ... */;

$seconds = 1;

while ($waiting) {
    Sleep::for($seconds++)->seconds();

    $waiting = /* ... */;
}
```

Typically, testing this code would take *at least* one second. Luckily, the `Sleep` class allows us to "fake" sleeping so that our test suite stays fast:

```
public function test_it_waits_until_ready()
{
    Sleep::fake();

    // ...
}
```

When faking the `Sleep` class, the actual execution pause is by-passed, leading to a substantially faster test.

Once the `Sleep` class has been faked, it is possible to make assertions against the expected "sleeps" that should have occurred. To illustrate this, let's imagine we are testing code that pauses execution three times, with each pause increasing by a single second. Using the `assertSequence` method, we can assert that our code "slept" for the proper amount of time while keeping our test fast:

```
public function test_it_checks_if_ready_four_times()
{
    Sleep::fake();

    // ...

    Sleep::assertSequence([
        Sleep::for(1)->second(),
        Sleep::for(2)->seconds(),
        Sleep::for(3)->seconds(),
    ]);
}
```

Of course, the `Sleep` class offers a variety of other assertions you may use when testing:

```

use Carbon\CarbonInterval as Duration;
use Illuminate\Support\Sleep;

// Assert that sleep was called 3 times...
Sleep::assertSleptTimes(3);

// Assert against the duration of sleep...
Sleep::assertSlept(function (Duration $duration): bool {
    return /* ... */;
}, times: 1);

// Assert that the Sleep class was never invoked...
Sleep::assertNeverSlept();

// Assert that, even if Sleep was called, no execution paused occurred...
Sleep::assertInsomniac();

```

Sometimes it may be useful to perform an action whenever a fake sleep occurs in your application code. To achieve this, you may provide a callback to the `whenFakingSleep` method. In the following example, we use Laravel's [time manipulation helpers](#) to instantly progress time by the duration of each sleep:

```

use Carbon\CarbonInterval as Duration;

$this->freezeTime();

Sleep::fake();

Sleep::whenFakingSleep(function (Duration $duration) {
    // Progress time when faking sleep...
    $this->travel($duration->totalMilliseconds())->milliseconds();
});

```

Laravel uses the `Sleep` class internally whenever it is pausing execution. For example, the `retry` helper uses the `Sleep` class when sleeping, allowing for improved testability when using that helper.

HTTP Client

- [Introduction](#)
- [Making Requests](#)
 - [Request Data](#)
 - [Headers](#)
 - [Authentication](#)
 - [Timeout](#)
 - [Retries](#)
 - [Error Handling](#)
 - [Guzzle Middleware](#)
 - [Guzzle Options](#)
- [Concurrent Requests](#)
- [Macros](#)
- [Testing](#)
 - [Faking Responses](#)
 - [Inspecting Requests](#)
 - [Preventing Stray Requests](#)
- [Events](#)

Introduction

Laravel provides an expressive, minimal API around the [Guzzle HTTP client](#), allowing you to quickly make outgoing HTTP requests to communicate with other web applications. Laravel's wrapper around Guzzle is focused on its most common use cases and a wonderful developer experience.

Before getting started, you should ensure that you have installed the Guzzle package as a dependency of your application. By default, Laravel automatically includes this dependency. However, if you have previously removed the package, you may install it again via Composer:

```
composer require guzzlehttp/guzzle
```

Making Requests

To make requests, you may use the `head`, `get`, `post`, `put`, `patch`, and `delete` methods provided by the `Http` facade. First, let's examine how to make a basic `GET` request to another URL:

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

The `get` method returns an instance of `Illuminate\Http\Client\Response`, which provides a variety of methods that may be used to inspect the response:

```
$response->body() : string;
$response->json($key = null, $default = null) : array|mixed;
$response->object() : object;
$response->collect($key = null) : Illuminate\Support\Collection;
$response->status() : int;
$response->successful() : bool;
$response->redirect(): bool;
$response->failed() : bool;
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

The `Illuminate\Http\Client\Response` object also implements the PHP `ArrayAccess` interface, allowing you to access JSON response data directly on the response:

```
return Http::get('http://example.com/users/1')['name'];
```

In addition to the response methods listed above, the following methods may be used to determine if the response has a given status code:

```
$response->ok() : bool; // 200 OK
$response->created() : bool; // 201 Created
$response->accepted() : bool; // 202 Accepted
$response->noContent() : bool; // 204 No Content
$response->movedPermanently() : bool; // 301 Moved Permanently
$response->found() : bool; // 302 Found
$response->badRequest() : bool; // 400 Bad Request
$response->unauthorized() : bool; // 401 Unauthorized
$response->paymentRequired() : bool; // 402 Payment Required
$response->forbidden() : bool; // 403 Forbidden
$response->notFound() : bool; // 404 Not Found
$response->requestTimeout() : bool; // 408 Request Timeout
$response->conflict() : bool; // 409 Conflict
$response->unprocessableEntity() : bool; // 422 Unprocessable Entity
$response->tooManyRequests() : bool; // 429 Too Many Requests
$response->serverError() : bool; // 500 Internal Server Error
```

URI Templates

The HTTP client also allows you to construct request URLs using the [URI template specification](#). To define the URL parameters that can be expanded by your URI template, you may use the `withUrlParameters` method:

```
Http::withUrlParameters([
    'endpoint' => 'https://laravel.com',
    'page' => 'docs',
    'version' => '9.x',
    'topic' => 'validation',
])->get('{+endpoint}/{page}/{version}/{topic}');
```

Dumping Requests

If you would like to dump the outgoing request instance before it is sent and terminate the script's execution, you may add the `dd` method to the beginning of your request definition:

```
return Http::dd()->get('http://example.com');
```

Request Data

Of course, it is common when making `POST`, `PUT`, and `PATCH` requests to send additional data with your request, so these methods accept an array of data as their second argument. By default, data will be sent using the `application/json` content type:

```
use Illuminate\Support\Facades\Http;

$response = Http::post('http://example.com/users', [
    'name' => 'Steve',
    'role' => 'Network Administrator',
]);
```

GET Request Query Parameters

When making `GET` requests, you may either append a query string to the URL directly or pass an array of key / value pairs as the second argument to the `get` method:

```
$response = Http::get('http://example.com/users', [
    'name' => 'Taylor',
    'page' => 1,
]);
```

Alternatively, the `withQueryParameters` method may be used:

```
Http::retry(3, 100)->withQueryParameters([
    'name' => 'Taylor',
    'page' => 1,
])->get('http://example.com/users')
```

Sending Form URL Encoded Requests

If you would like to send data using the `application/x-www-form-urlencoded` content type, you should call the `asForm` method before making your request:

```
$response = Http::asForm()->post('http://example.com/users', [
    'name' => 'Sara',
    'role' => 'Privacy Consultant',
]);
```

Sending a Raw Request Body

You may use the `withBody` method if you would like to provide a raw request body when making a request. The content type may be provided via the method's second argument:

```
$response = Http::withBody(
    base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

Multi-Part Requests

If you would like to send files as multi-part requests, you should call the `attach` method before making your request. This method accepts the name of the file and its contents. If needed, you may provide a third argument which will be considered the file's filename, while a fourth argument may be used to provide headers associated with the file:

```
$response = Http::attach(
    'attachment', file_get_contents('photo.jpg'), 'photo.jpg', ['Content-Type' => 'image/jpeg'])
->post('http://example.com/attachments');
```

Instead of passing the raw contents of a file, you may pass a stream resource:

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
    'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

Headers

Headers may be added to requests using the `withHeaders` method. This `withHeaders` method accepts an array of key / value pairs:

```
$response = Http::withHeaders([
    'X-First' => 'foo',
    'X-Second' => 'bar'
])->post('http://example.com/users', [
    'name' => 'Taylor',
]);
```

You may use the `accept` method to specify the content type that your application is expecting in response to your request:

```
$response = Http::accept('application/json')->get('http://example.com/users');
```

For convenience, you may use the `acceptJson` method to quickly specify that your application expects the `application/json` content type in response to your request:

```
$response = Http::acceptJson()->get('http://example.com/users');
```

The `withHeaders` method merges new headers into the request's existing headers. If needed, you may replace all of the headers entirely using the `replaceHeaders` method:

```
$response = Http::withHeaders([
    'X-Original' => 'foo',
])->replaceHeaders([
    'X-Replacement' => 'bar',
])->post('http://example.com/users', [
    'name' => 'Taylor',
]);
```

Authentication

You may specify basic and digest authentication credentials using the `withBasicAuth` and `withDigestAuth` methods, respectively:

```
// Basic authentication...
$response = Http::withBasicAuth('taylor@laravel.com', 'secret')->post(/* ... */);

// Digest authentication...
$response = Http::withDigestAuth('taylor@laravel.com', 'secret')->post(/* ... */);
```

Bearer Tokens

If you would like to quickly add a bearer token to the request's `Authorization` header, you may use the `withToken` method:

```
$response = Http::withToken('token')->post(/* ... */);
```

Timeout

The `timeout` method may be used to specify the maximum number of seconds to wait for a response. By default, the HTTP client will timeout after 30 seconds:

```
$response = Http::timeout(3)->get(/* ... */);
```

If the given timeout is exceeded, an instance of `Illuminate\Http\Client\ConnectionException` will be thrown.

You may specify the maximum number of seconds to wait while trying to connect to a server using the `connectTimeout` method:

```
$response = Http::connectTimeout(3)->get(/* ... */);
```

Retries

If you would like the HTTP client to automatically retry the request if a client or server error occurs, you may use the `retry` method. The `retry` method accepts the maximum number of times the request should be attempted and the number of milliseconds that Laravel should wait in between attempts:

```
$response = Http::retry(3, 100)->post(/* ... */);
```

If you would like to manually calculate the number of milliseconds to sleep between attempts, you may pass a closure as the second argument to the `retry` method:

```
use Exception;

$response = Http::retry(3, function (int $attempt, Exception $exception) {
    return $attempt * 100;
})->post(/* ... */);
```

For convenience, you may also provide an array as the first argument to the `retry` method. This array will be used to determine how many milliseconds to sleep between subsequent attempts:

```
$response = Http::retry([100, 200])->post(/* ... */);
```

If needed, you may pass a third argument to the `retry` method. The third argument should be a callable that determines if the retries should actually be attempted. For example, you may wish to only retry the request if the initial request encounters an `ConnectionException`:

```
use Exception;
use Illuminate\Http\Client\PendingRequest;

$response = Http::retry(3, 100, function (Exception $exception, PendingRequest $request) {
    return $exception instanceof ConnectionException;
})->post(/* ... */);
```

If a request attempt fails, you may wish to make a change to the request before a new attempt is made. You can achieve this by modifying the request argument provided to the callable you provided to the `retry` method. For example, you might want to retry the request with a new authorization token if the first attempt returned an authentication error:

```
use Exception;
use Illuminate\Http\Client\PendingRequest;
use Illuminate\Http\Client\RequestException;

$response = Http::withToken($this->getToken())->retry(2, 0, function (Exception $exception,
PendingRequest $request) {
    if (! $exception instanceof RequestException || $exception->response->status() !== 401) {
        return false;
    }

    $request->withToken($this->getNewToken());

    return true;
})->post(/* ... */);
```

If all of the requests fail, an instance of `Illuminate\Http\Client\RequestException` will be thrown. If you would like to disable this behavior, you may provide a `throw` argument with a value of `false`. When disabled, the last response received by the client will be returned after all retries have been attempted:

```
$response = Http::retry(3, 100, throw: false)->post(/* ... */);
```



If all of the requests fail because of a connection issue, a `Illuminate\Http\Client\ConnectionException` will still be thrown even when the `throw` argument is set to `false`.

Error Handling

Unlike Guzzle's default behavior, Laravel's HTTP client wrapper does not throw exceptions on client or server errors (`400` and `500` level responses from servers). You may determine if one of these errors was returned using the `successful`, `clientError`, or `serverError` methods:

```
// Determine if the status code is >= 200 and < 300...
$response->successful();

// Determine if the status code is >= 400...
$response->failed();

// Determine if the response has a 400 level status code...
$response->clientError();

// Determine if the response has a 500 level status code...
$response->serverError();

// Immediately execute the given callback if there was a client or server error...
$response->onError(callable $callback);
```

Throwing Exceptions

If you have a response instance and would like to throw an instance of `Illuminate\Http\Client\RequestException` if the response status code indicates a client or server error, you may use the `throw` or `throwIf` methods:

```
use Illuminate\Http\Client\Response;

$response = Http::post(/* ... */);

// Throw an exception if a client or server error occurred...
$response->throw();

// Throw an exception if an error occurred and the given condition is true...
$response->throwIf($condition);

// Throw an exception if an error occurred and the given closure resolves to true...
$response->throwIf(fn (Response $response) => true);

// Throw an exception if an error occurred and the given condition is false...
$response->throwUnless($condition);

// Throw an exception if an error occurred and the given closure resolves to false...
$response->throwUnless(fn (Response $response) => false);

// Throw an exception if the response has a specific status code...
$response->throwIfStatus(403);

// Throw an exception unless the response has a specific status code...
$response->throwUnlessStatus(200);

return $response['user']['id'];
```

The `Illuminate\Http\Client\RequestException` instance has a public `$response` property which will allow you to inspect the returned response.

The `throw` method returns the response instance if no error occurred, allowing you to chain other operations onto the `throw` method:

```
return Http::post(/* ... */)->throw()->json();
```

If you would like to perform some additional logic before the exception is thrown, you may pass a closure to the `throw` method. The exception will be thrown automatically after the closure is invoked, so you do not need to re-throw the exception from within the closure:

```
use Illuminate\Http\Client\Response;
use Illuminate\Http\Client\RequestException;

return Http::post(/* ... */)->throw(function (Response $response, RequestException $e) {
    // ...
})->json();
```

Guzzle Middleware

Since Laravel's HTTP client is powered by Guzzle, you may take advantage of [Guzzle Middleware](#) to manipulate the outgoing request or inspect the incoming response. To manipulate the outgoing request, register a Guzzle middleware via the `withRequestMiddleware` method:

```
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\RequestInterface;

$response = Http::withRequestMiddleware(
    function (RequestInterface $request) {
        return $request->withHeader('X-Example', 'Value');
    }
)->get('http://example.com');
```

Likewise, you can inspect the incoming HTTP response by registering a middleware via the `withResponseMiddleware` method:

```
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\ResponseInterface;

$response = Http::withResponseMiddleware(
    function (ResponseInterface $response) {
        $header = $response->getHeader('X-Example');

        // ...

        return $response;
    }
)->get('http://example.com');
```

Global Middleware

Sometimes, you may want to register a middleware that applies to every outgoing request and incoming response. To accomplish this, you may use the `globalRequestMiddleware` and `globalResponseMiddleware` methods. Typically, these methods should be invoked in the `boot` method of your application's [AppServiceProvider](#):

```
use Illuminate\Support\Facades\Http;

Http::globalRequestMiddleware(fn ($request) => $request->withHeader(
    'User-Agent', 'Example Application/1.0'
));

Http::globalResponseMiddleware(fn ($response) => $response->withHeader(
    'X-Finished-At', now()->toDateTimeString()
));
```

Guzzle Options

You may specify additional [Guzzle request options](#) using the `withOptions` method. The `withOptions` method accepts an array of key / value pairs:

```
$response = Http::withOptions([
    'debug' => true,
])->get('http://example.com/users');
```

Concurrent Requests

Sometimes, you may wish to make multiple HTTP requests concurrently. In other words, you want several requests to be dispatched at the same time instead of issuing the requests sequentially. This can lead to substantial performance improvements when interacting with slow HTTP APIs.

Thankfully, you may accomplish this using the `pool` method. The `pool` method accepts a closure which receives an [Illuminate\Http\Client\Pool](#) instance, allowing you to easily add requests to the request pool for dispatching:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->get('http://localhost/first'),
    $pool->get('http://localhost/second'),
    $pool->get('http://localhost/third'),
]);

return $responses[0]->ok() &&
    $responses[1]->ok() &&
    $responses[2]->ok();
```

As you can see, each response instance can be accessed based on the order it was added to the pool. If you wish, you can name the requests using the `as` method, which allows you to access the corresponding responses by name:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->as('first')->get('http://localhost/first'),
    $pool->as('second')->get('http://localhost/second'),
    $pool->as('third')->get('http://localhost/third'),
]);

return $responses['first']->ok();
```

Customizing Concurrent Requests

The `pool` method cannot be chained with other HTTP client methods such as the `withHeaders` or `middleware` methods. If you want to apply custom headers or middleware to pooled requests, you should configure those options on each request in the pool:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$headers = [
    'X-Example' => 'example',
];

$responses = Http::pool(fn (Pool $pool) => [
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
]);
```

Macros

The Laravel HTTP client allows you to define "macros", which can serve as a fluent, expressive mechanism to configure common request paths and headers when interacting with services throughout your application. To get started, you may define the macro within the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use Illuminate\Support\Facades\Http;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Http::macro('github', function () {
        return Http::withHeaders([
            'X-Example' => 'example',
        ])->baseUrl('https://github.com');
    });
}
```

Once your macro has been configured, you may invoke it from anywhere in your application to create a pending request with the specified configuration:

```
$response = Http::github()->get('/');
```

Testing

Many Laravel services provide functionality to help you easily and expressively write tests, and Laravel's HTTP client is no exception. The `Http` facade's `fake` method allows you to instruct the HTTP client to return stubbed / dummy responses when requests are made.

Faking Responses

For example, to instruct the HTTP client to return empty, `200` status code responses for every request, you may call the `fake` method with no arguments:

```
use Illuminate\Support\Facades\Http;

Http::fake();

$response = Http::post(/* ... */);
```

Faking Specific URLs

Alternatively, you may pass an array to the `fake` method. The array's keys should represent URL patterns that you wish to fake and their associated responses. The `*` character may be used as a wildcard character. Any requests made to URLs that have not been faked will actually be executed. You may use the `Http` facade's `response` method to construct stub / fake responses for these endpoints:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

    // Stub a string response for Google endpoints...
    'google.com/*' => Http::response('Hello World', 200, $headers),
]);
```

If you would like to specify a fallback URL pattern that will stub all unmatched URLs, you may use a single `*` character:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, ['Headers']),

    // Stub a string response for all other endpoints...
    '*' => Http::response('Hello World', 200, ['Headers']),
]);
```

Faking Response Sequences

Sometimes you may need to specify that a single URL should return a series of fake responses in a specific order. You may accomplish this using the `Http::sequence` method to build the responses:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
        ->push('Hello World', 200)
        ->push(['foo' => 'bar'], 200)
        ->pushStatus(404),
]);
});
```

When all the responses in a response sequence have been consumed, any further requests will cause the response sequence to throw an exception. If you would like to specify a default response that should be returned when a sequence is empty, you may use the `whenEmpty` method:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
        ->push('Hello World', 200)
        ->push(['foo' => 'bar'], 200)
        ->whenEmpty(Http::response()),
]);
});
```

If you would like to fake a sequence of responses but do not need to specify a specific URL pattern that should be faked, you may use the `Http::fakeSequence` method:

```
Http::fakeSequence()
    ->push('Hello World', 200)
    ->whenEmpty(Http::response());
```

Fake Callback

If you require more complicated logic to determine what responses to return for certain endpoints, you may pass a closure to the `fake` method. This closure will receive an instance of `Illuminate\Http\Client\Request` as well as an array of options. The closure should return a response instance. Within your closure, you may perform whatever logic is necessary to determine what type of response to return:

```
use Illuminate\Http\Client\Request;

Http::fake(function (Request $request, array $options) {
    return Http::response('Hello World', 200);
});
```

Preventing Stray Requests

If you would like to ensure that all requests sent via the HTTP client have been faked throughout your individual test or complete test suite, you can call the `preventStrayRequests` method. After calling this method, any requests that do not have a corresponding fake response will throw an exception rather than making the actual HTTP request:

```
use Illuminate\Support\Facades\Http;

Http::preventStrayRequests();

Http::fake([
    'github.com/*' => Http::response('ok'),
]);

// An "ok" response is returned...
Http::get('https://github.com/laravel/framework');

// An exception is thrown...
Http::get('https://laravel.com');
```

Inspecting Requests

When faking responses, you may occasionally wish to inspect the requests the client receives in order to make sure your application is sending the correct data or headers. You may accomplish this by calling the `Http::assertSent` method after calling `Http::fake`.

The `assertSent` method accepts a closure which will receive an `Illuminate\Http\Client\Request` instance and should return a boolean value indicating if the request matches your expectations. In order for the test to pass, at least one request must have been issued matching the given expectations:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
    'X-First' => 'foo',
])->post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
    return $request->hasHeader('X-First', 'foo') &&
        $request->url() == 'http://example.com/users' &&
        $request['name'] == 'Taylor' &&
        $request['role'] == 'Developer';
});
```

If needed, you may assert that a specific request was not sent using the `assertNotSent` method:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

You may use the `assertSentCount` method to assert how many requests were "sent" during the test:

```
Http::fake();

Http::assertSentCount(5);
```

Or, you may use the `assertNothingSent` method to assert that no requests were sent during the test:

```
Http::fake();

Http::assertNothingSent();
```

Recording Requests / Responses

You may use the `recorded` method to gather all requests and their corresponding responses. The `recorded` method returns a collection of arrays that contains instances of `Illuminate\Http\Client\Request` and `Illuminate\Http\Client\Response`:

```
Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded();

[$request, $response] = $recorded[0];
```

Additionally, the `recorded` method accepts a closure which will receive an instance of `Illuminate\Http\Client\Request` and `Illuminate\Http\Client\Response` and may be used to filter request / response pairs based on your expectations:

```

use Illuminate\Http\Client\Request;
use Illuminate\Http\Client\Response;

Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded(function (Request $request, Response $response) {
    return $request->url() !== 'https://laravel.com' &&
        $response->successful();
});

```

Events

Laravel fires three events during the process of sending HTTP requests. The `RequestSending` event is fired prior to a request being sent, while the `ResponseReceived` event is fired after a response is received for a given request. The `ConnectionFailed` event is fired if no response is received for a given request.

The `RequestSending` and `ConnectionFailed` events both contain a public `$request` property that you may use to inspect the `Illuminate\Http\Client\Request` instance. Likewise, the `ResponseReceived` event contains a `$request` property as well as a `$response` property which may be used to inspect the `Illuminate\Http\Client\Response` instance. You may register event listeners for this event in your `App\Providers\EventServiceProvider` service provider:

```

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Http\Client\Events\RequestSending' => [
        'App\Listeners\LogRequestSending',
    ],
    'Illuminate\Http\Client\Events\ResponseReceived' => [
        'App\Listeners\LogResponseReceived',
    ],
    'Illuminate\Http\Client\Events\ConnectionFailed' => [
        'App\Listeners\LogConnectionFailed',
    ],
];

```

Localization

- [Introduction](#)
 - [Publishing the Language Files](#)
 - [Configuring the Locale](#)
 - [Pluralization Language](#)
- [Defining Translation Strings](#)
 - [Using Short Keys](#)
 - [Using Translation Strings as Keys](#)
- [Retrieving Translation Strings](#)
 - [Replacing Parameters in Translation Strings](#)
 - [Pluralization](#)
- [Overriding Package Language Files](#)

Introduction



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

Laravel's localization features provide a convenient way to retrieve strings in various languages, allowing you to easily support multiple languages within your application.

Laravel provides two ways to manage translation strings. First, language strings may be stored in files within the application's `lang` directory. Within this directory, there may be subdirectories for each language supported by the application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/lang
  /en
    messages.php
  /es
    messages.php
```

Or, translation strings may be defined within JSON files that are placed within the `lang` directory. When taking this approach, each language supported by your application would have a corresponding JSON file within this directory. This approach is recommended for applications that have a large number of translatable strings:

```
/lang
  en.json
  es.json
```

We'll discuss each approach to managing translation strings within this documentation.

Publishing the Language Files

By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files or create your own, you should scaffold the `lang` directory via the `lang:publish` Artisan command. The

`lang:publish` command will create the `lang` directory in your application and publish the default set of language files used by Laravel:

```
php artisan lang:publish
```

Configuring the Locale

The default language for your application is stored in the `config/app.php` configuration file's `locale` configuration option. You are free to modify this value to suit the needs of your application.

You may modify the default language for a single HTTP request at runtime using the `setLocale` method provided by the `App` facade:

```
use Illuminate\Support\Facades\App;

Route::get('/greeting/{locale}', function (string $locale) {
    if (! in_array($locale, ['en', 'es', 'fr'])) {
        abort(400);
    }

    App::setLocale($locale);

    // ...
});
```

You may configure a "fallback language", which will be used when the active language does not contain a given translation string. Like the default language, the fallback language is also configured in the `config/app.php` configuration file:

```
'fallback_locale' => 'en',
```

Determining the Current Locale

You may use the `currentLocale` and `isLocale` methods on the `App` facade to determine the current locale or check if the locale is a given value:

```
use Illuminate\Support\Facades\App;

$locale = App::currentLocale();

if (App::isLocale('en')) {
    // ...
}
```

Pluralization Language

You may instruct Laravel's "pluralizer", which is used by Eloquent and other portions of the framework to convert singular strings to plural strings, to use a language other than English. This may be accomplished by invoking the `useLanguage` method within the `boot` method of one of your application's service providers. The pluralizer's currently supported languages are: `french`, `norwegian-bokmal`, `portuguese`, `spanish`, and `turkish`:

```
use Illuminate\Support\Pluralizer;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Pluralizer::useLanguage('spanish');

    // ...
}
```



If you customize the pluralizer's language, you should explicitly define your Eloquent model's [table names](#).

Defining Translation Strings

Using Short Keys

Typically, translation strings are stored in files within the `lang` directory. Within this directory, there should be a subdirectory for each language supported by your application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/lang
  /en
    messages.php
  /es
    messages.php
```

All language files return an array of keyed strings. For example:

```
<?php

// lang/en/messages.php

return [
    'welcome' => 'Welcome to our application!',
];
```



For languages that differ by territory, you should name the language directories according to the ISO 15897. For example, "en_GB" should be used for British English rather than "en-gb".

Using Translation Strings as Keys

For applications with a large number of translatable strings, defining every string with a "short key" can become confusing when referencing the keys in your views and it is cumbersome to continually invent keys for every translation string supported by your application.

For this reason, Laravel also provides support for defining translation strings using the "default" translation of the string as the key. Language files that use translation strings as keys are stored as JSON files in the `lang` directory. For example, if your application has a Spanish translation, you should create a `lang/es.json` file:

```
{
    "I love programming.": "Me encanta programar."
}
```

Key / File Conflicts

You should not define translation string keys that conflict with other translation filenames. For example, translating `__('Action')` for the "NL" locale while a `nl/action.php` file exists but a `nl.json` file does not exist will result in the translator returning the entire contents of `nl/action.php`.

Retrieving Translation Strings

You may retrieve translation strings from your language files using the `__` helper function. If you are using "short keys" to define your translation strings, you should pass the file that contains the key and the key itself to the `__` function using "dot" syntax. For example, let's retrieve the `welcome` translation string from the `lang/en/messages.php` language file:

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

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

If you are using your default translation strings as your translation keys, you should pass the default translation of your string to the `__` function;

```
echo __('I love programming.');
```

Again, if the translation string does not exist, the `__` function will return the translation string key that it was given.

If you are using the Blade templating engine, you may use the `{{ }}` echo syntax to display the translation string:

```
{{ __('messages.welcome') }}
```

Replacing Parameters in Translation Strings

If you wish, you may define placeholders in your translation strings. All placeholders are prefixed with a `:`. For example, you may define a welcome message with a placeholder name:

```
'welcome' => 'Welcome, :name',
```

To replace the placeholders when retrieving a translation string, you may pass an array of replacements as the second argument to the `__` function:

```
echo ___('messages.welcome', ['name' => 'dayle']);
```

If your placeholder contains all capital letters, or only has its first letter capitalized, the translated value will be capitalized accordingly:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

Object Replacement Formatting

If you attempt to provide an object as a translation placeholder, the object's `__toString` method will be invoked. The `__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, Laravel allows you to register a custom formatting handler for that particular type of object. To accomplish this, you should invoke the translator's `stringable` method. The `stringable` method accepts a closure, which should type-hint the type of object that it is responsible for formatting. Typically, the `stringable` method should be invoked within the `boot` method of your application's `AppServiceProvider` class:

```
use Illuminate\Support\Facades\Lang;
use Money\Money;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Lang::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization; however, Laravel can help you translate strings differently based on pluralization rules that you define. Using a `|` character, you may distinguish singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

Of course, pluralization is also supported when using [translation strings as keys](#):

```
{
    "There is one apple|There are many apples": "Hay una manzana|Hay muchas manzanas"
}
```

You may even create more complex pluralization rules which specify translation strings for multiple ranges of values:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

After defining a translation string that has pluralization options, you may use the `trans_choice` function to retrieve the line for a given "count". In this example, since the count is greater than one, the plural form of the translation string is returned:

```
echo trans_choice('messages.apples', 10);
```

You may also define placeholder attributes in pluralization strings. These placeholders may be replaced by passing an array as the third argument to the `trans_choice` function:

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',  
echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

If you would like to display the integer value that was passed to the `trans_choice` function, you may use the built-in `:count` placeholder:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

Overriding Package Language Files

Some packages may ship with their own language files. Instead of changing the package's core files to tweak these lines, you may override them by placing files in the `lang/vendor/{package}/{locale}` directory.

So, for example, if you need to override the English translation strings in `messages.php` for a package named `skyrim/hearthfire`, you should place a language file at: `lang/vendor/hearthfire/en/messages.php`. Within this file, you should only define the translation strings you wish to override. Any translation strings you don't override will still be loaded from the package's original language files.

Mail

- [Introduction](#)
 - [Configuration](#)
 - [Driver Prerequisites](#)
 - [Failover Configuration](#)
 - [Round Robin Configuration](#)
- [Generating Mailables](#)
- [Writing Mailables](#)
 - [Configuring the Sender](#)
 - [Configuring the View](#)
 - [View Data](#)
 - [Attachments](#)
 - [Inline Attachments](#)
 - [Attachable Objects](#)
 - [Headers](#)
 - [Tags and Metadata](#)
 - [Customizing the Symfony Message](#)
- [Markdown Mailables](#)
 - [Generating Markdown Mailables](#)
 - [Writing Markdown Messages](#)
 - [Customizing the Components](#)
- [Sending Mail](#)
 - [Queueing Mail](#)
- [Rendering Mailables](#)
 - [Previewing Mailables in the Browser](#)
- [Localizing Mailables](#)
- [Testing](#)
 - [Testing Mailable Content](#)
 - [Testing Mailable Sending](#)
- [Mail and Local Development](#)
- [Events](#)
- [Custom Transports](#)
 - [Additional Symfony Transports](#)

Introduction

Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular [Symfony Mailer](#) component. Laravel and Symfony Mailer provide drivers for sending email via SMTP, Mailgun, Postmark, Amazon SES, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Configuration

Laravel's email services may be configured via your application's `config/mail.php` configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.

Within your `mail` configuration file, you will find a `mailers` configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the `default` configuration value determines which mailer will be used by default when your application needs to send an email message.

Driver / Transport Prerequisites

The API based drivers such as Mailgun, Postmark, and MailerSend are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers.

Mailgun Driver

To use the Mailgun driver, install Symfony's Mailgun Mailer transport via Composer:

```
composer require symfony/mailgun-mailer symfony/http-client
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `mailgun`. After configuring your application's default mailer, verify that your `config/services.php` configuration file contains the following options:

```
'mailgun' => [
    'transport' => 'mailgun',
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
],
```

If you are not using the United States [Mailgun region](#), you may define your region's endpoint in the `services` configuration file:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
],
```

Postmark Driver

To use the Postmark driver, install Symfony's Postmark Mailer transport via Composer:

```
composer require symfony/postmark-mailer symfony/http-client
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `postmark`. After configuring your application's default mailer, verify that your `config/services.php` configuration file contains the following options:

```
'postmark' => [
    'token' => env('POSTMARK_TOKEN'),
],
```

If you would like to specify the Postmark message stream that should be used by a given mailer, you may add the `message_stream_id` configuration option to the mailer's configuration array. This configuration array can be found in your application's `config/mail.php` configuration file:

```
'postmark' => [
    'transport' => 'postmark',
    'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
],
```

This way you are also able to set up multiple Postmark mailers with different message streams.

SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library via the Composer package manager:

```
composer require aws/aws-sdk-php
```

Next, set the `default` option in your `config/mail.php` configuration file to `ses` and verify that your `config/services.php` configuration file contains the following options:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

To utilize AWS [temporary credentials](#) via a session token, you may add a `token` key to your application's SES configuration:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'token' => env('AWS_SESSION_TOKEN'),
],
```

If you would like to define [additional options](#) that Laravel should pass to the AWS SDK's `SendEmail` method when sending an email, you may define an `options` array within your `ses` configuration:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'EmailTags' => [
            ['Name' => 'foo', 'Value' => 'bar'],
        ],
    ],
],
```

MailerSend Driver

[MailerSend](#), a transactional email and SMS service, maintains their own API based mail driver for Laravel. The package containing the driver may be installed via the Composer package manager:

```
composer require mailersend/laravel-driver
```

Once the package is installed, add the `MAILERSEND_API_KEY` environment variable to your application's `.env` file. In addition, the `MAIL_MAILER` environment variable should be defined as `mailersend`:

```
MAIL_MAILER=mailersend
MAIL_FROM_ADDRESS=app@yourdomain.com
MAIL_FROM_NAME="App Name"

MAILERSEND_API_KEY=your-api-key
```

To learn more about MailerSend, including how to use hosted templates, consult the [MailerSend driver documentation](#).

Failover Configuration

Sometimes, an external service you have configured to send your application's mail may be down. In these cases, it can be useful to define one or more backup mail delivery configurations that will be used in case your primary delivery driver is down.

To accomplish this, you should define a mailer within your application's `mail` configuration file that uses the `failover` transport. The configuration array for your application's `failover` mailer should contain an array of `mailers` that reference the order in which configured mailers should be chosen for delivery:

```
'mailers' => [
    'failover' => [
        'transport' => 'failover',
        'mailers' => [
            'postmark',
            'mailgun',
            'sendmail',
        ],
    ],
    // ...
],
```

Once your failover mailer has been defined, you should set this mailer as the default mailer used by your application by specifying its name as the value of the `default` configuration key within your application's `mail` configuration file:

```
'default' => env('MAIL_MAILER', 'failover'),
```

Round Robin Configuration

The `roundrobin` transport allows you to distribute your mailing workload across multiple mailers. To get started, define a mailer within your application's `mail` configuration file that uses the `roundrobin` transport. The configuration array for your application's `roundrobin` mailer should contain an array of `mailers` that reference which configured mailers should be used for delivery:

```
'mailers' => [
    'roundrobin' => [
        'transport' => 'roundrobin',
        'mailers' => [
            'ses',
            'postmark',
        ],
    ],
    // ...
],
```

Once your round robin mailer has been defined, you should set this mailer as the default mailer used by your application by specifying its name as the value of the `default` configuration key within your application's `mail` configuration file:

```
'default' => env('MAIL_MAILER', 'roundrobin'),
```

The round robin transport selects a random mailer from the list of configured mailers and then switches to the next available mailer for each subsequent email. In contrast to `failover` transport, which helps to achieve *high availability*, the `roundrobin` transport provides *load balancing*.

Generating Mailables

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the `app/Mail` directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped
```

Writing Mailables

Once you have generated a mailable class, open it up so we can explore its contents. Mailable class configuration is done in several methods, including the `envelope`, `content`, and `attachments` methods.

The `envelope` method returns an `Illuminate\Mail\Mailables\Envelope` object that defines the subject and, sometimes, the recipients of the message. The `content` method returns an `Illuminate\Mail\Mailables\Content` object that defines the Blade template that will be used to generate the message content.

Configuring the Sender

Using the Envelope

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may specify the "from" address on your message's envelope:

```
use Illuminate\Mail\Mailables\Address;
use Illuminate\Mail\Mailables\Envelope;

/**
 * Get the message envelope.
 */
public function envelope(): Envelope
{
    return new Envelope(
        from: new Address('jeffrey@example.com', 'Jeffrey Way'),
        subject: 'Order Shipped',
    );
}
```

If you would like, you may also specify a `replyTo` address:

```
return new Envelope(
    from: new Address('jeffrey@example.com', 'Jeffrey Way'),
    replyTo: [
        new Address('taylor@example.com', 'Taylor Otwell'),
    ],
    subject: 'Order Shipped',
);
```

Using a Global `from` Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to add it to each mailable class you generate. Instead, you may specify a global "from" address in your `config/mail.php` configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
],
```

In addition, you may define a global "reply_to" address within your `config/mail.php` configuration file:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

Configuring the View

Within a mailable class's `content` method, you may define the `view`, or which template should be used when rendering the email's contents. Since each email typically uses a [Blade template](#) to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

```
/**
 * Get the message content definition.
 */
public function content(): Content
{
    return new Content(
        view: 'mail.orders.shipped',
    );
}
```



You may wish to create a `resources/views/emails` directory to house all of your email templates; however, you are free to place them wherever you wish within your `resources/views` directory.

Plain Text Emails

If you would like to define a plain-text version of your email, you may specify the plain-text template when creating the message's `Content` definition. Like the `view` parameter, the `text` parameter should be a template name which will be used to render the contents of the email. You are free to define both an HTML and plain-text version of your message:

```
/**  
 * Get the message content definition.  
 */  
public function content(): Content  
{  
    return new Content(  
        view: 'mail.orders.shipped',  
        text: 'mail.orders.shipped-text'  
    );  
}
```

For clarity, the `html` parameter may be used as an alias of the `view` parameter:

```
return new Content(  
    html: 'mail.orders.shipped',  
    text: 'mail.orders.shipped-text'  
)
```

View Data

Via Public Properties

Typically, you will want to pass some data to your view that you can utilize when rendering the email's HTML. There are two ways you may make data available to your view. First, any public property defined on your mailable class will automatically be made available to the view. So, for example, you may pass data into your mailable class's constructor and set that data to public properties defined on the class:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct(
        public Order $order,
    ) {}

    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
        );
    }
}
```

Once the data has been set to a public property, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $order->price }}
</div>
```

Via the `with` Parameter:

If you would like to customize the format of your email's data before it is sent to the template, you may manually pass your data to the view via the `Content` definition's `with` parameter. Typically, you will still pass data via the mailable class's constructor; however, you should set this data to `protected` or `private` properties so the data is not automatically made available to the template:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct(
        protected Order $order,
    ) {}

    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
            with: [
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
            ],
        );
    }
}
```

Once the data has been passed to the `with` method, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $orderPrice }}
</div>
```

Attachments

To add attachments to an email, you will add attachments to the array returned by the message's `attachments` method. First, you may add an attachment by providing a file path to the `fromPath` method provided by the `Attachment` class:

```
use Illuminate\Mail\Mailables\Attachment;

/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file'),
    ];
}
```

When attaching files to a message, you may also specify the display name and / or MIME type for the attachment using the `as` and `withMime` methods:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf'),
    ];
}
```

Attaching Files From Disk

If you have stored a file on one of your [filesystem disks](#), you may attach it to the email using the `fromStorage` attachment method:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file'),
    ];
}
```

Of course, you may also specify the attachment's name and MIME type:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf'),
    ];
}
```

The `fromStorageDisk` method may be used if you need to specify a storage disk other than your default disk:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorageDisk('s3', '/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf'),
    ];
}
```

Raw Data Attachments

The `fromData` attachment method may be used to attach a raw string of bytes as an attachment. For example, you might use this method if you have generated a PDF in memory and want to attach it to the email without writing it to disk. The `fromData` method accepts a closure which resolves the raw data bytes as well as the name that the attachment should be assigned:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromData(fn () => $this->pdf, 'Report.pdf')
            ->withMime('application/pdf'),
    ];
}
```

Inline Attachments

Embedding inline images into your emails is typically cumbersome; however, Laravel provides a convenient way to attach images to your emails. To embed an inline image, use the `embed` method on the `$message` variable within your email

template. Laravel automatically makes the `$message` variable available to all of your email templates, so you don't need to worry about passing it in manually:

```
<body>
    Here is an image:

    
</body>
```



The `$message` variable is not available in plain-text message templates since plain-text messages do not utilize inline attachments.

Embedding Raw Data Attachments

If you already have a raw image data string you wish to embed into an email template, you may call the `embedData` method on the `$message` variable. When calling the `embedData` method, you will need to provide a filename that should be assigned to the embedded image:

```
<body>
    Here is an image from raw data:

    
</body>
```

Attachable Objects

While attaching files to messages via simple string paths is often sufficient, in many cases the attachable entities within your application are represented by classes. For example, if your application is attaching a photo to a message, your application may also have a `Photo` model that represents that photo. When that is the case, wouldn't it be convenient to simply pass the `Photo` model to the `attach` method? Attachable objects allow you to do just that.

To get started, implement the `Illuminate\Contracts\Mail\Attachable` interface on the object that will be attachable to messages. This interface dictates that your class defines a `toMailAttachment` method that returns an `Illuminate\Mail\Attachment` instance:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Mail\Attachable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Mail\Attachment;

class Photo extends Model implements Attachable
{
    /**
     * Get the attachable representation of the model.
     */
    public function toMailAttachment(): Attachment
    {
        return Attachment::fromPath('/path/to/file');
    }
}
```

Once you have defined your attachable object, you may return an instance of that object from the `attachments` method when building an email message:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [$this->photo];
}
```

Of course, attachment data may be stored on a remote file storage service such as Amazon S3. So, Laravel also allows you to generate attachment instances from data that is stored on one of your application's [filesystem disks](#):

```
// Create an attachment from a file on your default disk...
return Attachment::fromStorage($this->path);

// Create an attachment from a file on a specific disk...
return Attachment::fromStorageDisk('backblaze', $this->path);
```

In addition, you may create attachment instances via data that you have in memory. To accomplish this, provide a closure to the `fromData` method. The closure should return the raw data that represents the attachment:

```
return Attachment::fromData(fn () => $this->content, 'Photo Name');
```

Laravel also provides additional methods that you may use to customize your attachments. For example, you may use the `as` and `withMime` methods to customize the file's name and MIME type:

```
return Attachment::fromPath('/path/to/file')
    ->as('Photo Name')
    ->withMime('image/jpeg');
```

Headers

Sometimes you may need to attach additional headers to the outgoing message. For instance, you may need to set a custom `Message-Id` or other arbitrary text headers.

To accomplish this, define a `headers` method on your mailable. The `headers` method should return an `\Illuminate\Mail\Mailables\Headers` instance. This class accepts `messageId`, `references`, and `text` parameters. Of course, you may provide only the parameters you need for your particular message:

```
use Illuminate\Mail\Mailables\Headers;

/**
 * Get the message headers.
 */
public function headers(): Headers
{
    return new Headers(
        messageId: 'custom-message-id@example.com',
        references: ['previous-message@example.com'],
        text: [
            'X-Custom-Header' => 'Custom Value',
        ],
    );
}
```

Tags and Metadata

Some third-party email providers such as Mailgun and Postmark support message "tags" and "metadata", which may be used to group and track emails sent by your application. You may add tags and metadata to an email message via your `Envelope` definition:

```
use Illuminate\Mail\Mailables\Envelope;

/**
 * Get the message envelope.
 *
 * @return \Illuminate\Mail\Mailables\Envelope
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Order Shipped',
        tags: ['shipment'],
        metadata: [
            'order_id' => $this->order->id,
        ],
    );
}
```

If your application is using the Mailgun driver, you may consult Mailgun's documentation for more information on [tags](#) and [metadata](#). Likewise, the Postmark documentation may also be consulted for more information on their support for [tags](#) and [metadata](#).

If your application is using Amazon SES to send emails, you should use the `metadata` method to attach [SES "tags"](#) to the message.

Customizing the Symfony Message

Laravel's mail capabilities are powered by Symfony Mailer. Laravel allows you to register custom callbacks that will be invoked with the Symfony Message instance before sending the message. This gives you an opportunity to deeply customize the message before it is sent. To accomplish this, define a `using` parameter on your `Envelope` definition:

```
use Illuminate\Mail\Mailables\Envelope;
use Symfony\Component\Mime\Email;

/**
 * Get the message envelope.
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Order Shipped',
        using: [
            function (Email $message) {
                // ...
            },
        ],
    );
}
```

Markdown Mailables

Markdown mailable messages allow you to take advantage of the pre-built templates and components of [mail notifications](#) in your mailables. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating Markdown Mailables

To generate a mailable with a corresponding Markdown template, you may use the `--markdown` option of the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped --markdown=mail.orders.shipped
```

Then, when configuring the mailable `Content` definition within its `content` method, use the `markdown` parameter instead of the `view` parameter:

```
use Illuminate\Mail\Mailables\Content;

/**
 * Get the message content definition.
 */
public function content(): Content
{
    return new Content(
        markdown: 'mail.orders.shipped',
        with: [
            'url' => $this->orderUrl,
        ],
    );
}
```

Writing Markdown Messages

Markdown mailables use a combination of Blade components and Markdown syntax which allow you to easily construct mail messages while leveraging Laravel's pre-built email UI components:

```
<x-mail::message>
# Order Shipped

Your order has been shipped!

<x-mail::button :url="$url">
View Order
</x-mail::button>

Thanks,<br>
{{ config('app.name') }}
</x-mail::message>
```



Do not use excess indentation when writing Markdown emails. Per Markdown standards, Markdown parsers will render indented content as code blocks.

Button Component

The button component renders a centered button link. The component accepts two arguments, a `:url` and an optional `color`. Supported colors are `primary`, `success`, and `error`. You may add as many button components to a message as you wish:

```
<x-mail::button :url="$url" color="success">
View Order
</x-mail::button>
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the message. This allows you to draw attention to a given block of text:

```
<x-mail::panel>
This is the panel content.
</x-mail::panel>
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
<x-mail::table>
| Laravel      | Table          | Example   |
| ----- |:-----:| -----:|
| Col 2 is    | Centered      | $10       |
| Col 3 is    | Right-Aligned | $20       |
</x-mail::table>
```

Customizing the Components

You may export all of the Markdown mail components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

```
php artisan vendor:publish --tag=laravel-mail
```

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing the CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be converted to inline CSS styles within the HTML representations of your Markdown mail messages.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of your application's `config/mail.php` configuration file to match the name of your new theme.

To customize the theme for an individual mailable, you may set the `$theme` property of the mailable class to the name of the theme that should be used when sending that mailable.

Sending Mail

To send a message, use the `to` method on the `Mail facade`. The `to` method accepts an email address, a user instance, or a collection of users. If you pass an object or collection of objects, the mailer will automatically use their `email` and `name` properties when determining the email's recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // Ship the order...

        Mail::to($request->user())->send(new OrderShipped($order));

        return redirect('/orders');
    }
}
```

You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients by chaining their respective methods together:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

Looping Over Recipients

Occasionally, you may need to send a mailable to a list of recipients by iterating over an array of recipients / email addresses. However, since the `to` method appends email addresses to the mailable's list of recipients, each iteration through the loop will send another email to every previous recipient. Therefore, you should always re-create the mailable instance for each recipient:

```
foreach(['taylor@example.com', 'dries@example.com'] as $recipient) {
    Mail::to($recipient)->send(new OrderShipped($order));
}
```

Sending Mail via a Specific Mailer

By default, Laravel will send email using the mailer configured as the `default` mailer in your application's `mail` configuration file. However, you may use the `mailer` method to send a message using a specific mailer configuration:

```
Mail::mailer('postmark')
    ->to($request->user())
    ->send(new OrderShipped($order));
```

Queueing Mail

Queueing a Mail Message

Since sending email messages can negatively impact the response time of your application, many developers choose to queue email messages for background sending. Laravel makes this easy using its built-in [unified queue API](#). To queue a mail message, use the `queue` method on the `Mail` facade after specifying the message's recipients:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

This method will automatically take care of pushing a job onto the queue so the message is sent in the background. You will need to [configure your queues](#) before using this feature.

Delayed Message Queueing

If you wish to delay the delivery of a queued email message, you may use the `later` method. As its first argument, the `later` method accepts a `DateTime` instance indicating when the message should be sent:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

Pushing to Specific Queues

Since all mailable classes generated using the `make:mail` command make use of the `Illuminate\Bus\Queueable` trait, you may call the `onQueue` and `onConnection` methods on any mailable class instance, allowing you to specify the connection and queue name for the message:

```
$message = (new OrderShipped($order))
    ->onConnection('sq')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

Queueing by Default

If you have mailable classes that you want to always be queued, you may implement the `ShouldQueue` contract on the class. Now, even if you call the `send` method when mailing, the mailable will still be queued since it implements the contract:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    // ...
}
```

Queued Mailables and Database Transactions

When queued mailables are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your mailable depends on these models, unexpected errors can occur when the job that sends the queued mailable is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued mailable should be dispatched after all open database transactions have been committed by calling the `afterCommit` method when sending the mail message:

```
Mail::to($request->user())->send(
    new OrderShipped($order))->afterCommit()
);
```

Alternatively, you may call the `afterCommit` method from your mailable's constructor:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable implements ShouldQueue
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```



To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).

Rendering Mailables

Sometimes you may wish to capture the HTML content of a mailable without sending it. To accomplish this, you may call the `render` method of the mailable. This method will return the evaluated HTML content of the mailable as a string:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;

$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))->render();
```

Previewing Mailables in the Browser

When designing a mailable's template, it is convenient to quickly preview the rendered mailable in your browser like a typical Blade template. For this reason, Laravel allows you to return any mailable directly from a route closure or controller. When a mailable is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
Route::get('/mailable', function () {
    $invoice = App\Models\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

Localizing Mailables

Laravel allows you to send mailables in a locale other than the request's current locale, and will even remember this locale if the mail is queued.

To accomplish this, the `Mail` facade offers a `locale` method to set the desired language. The application will change into this locale when the mailable's template is being evaluated and then revert back to the previous locale when evaluation is complete:

```
Mail::to($request->user())->locale('es')->send(
    new OrderShipped($order)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on one or more of your models, you may instruct Laravel to use this stored locale when sending mail:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending mailables and notifications to the model. Therefore, there is no need to call the `locale` method when using this interface:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

Testing

Testing Mailable Content

Laravel provides a variety of methods for inspecting your mailable's structure. In addition, Laravel provides several convenient methods for testing that your mailable contains the content that you expect. These methods are:

```
assertSeeInHtml, assertDontSeeInHtml, assertSeeInOrderInHtml, assertSeeInText,  
assertDontSeeInText, assertSeeInOrderInText, assertHasAttachment, assertHasAttachedData,  
assertHasAttachmentFromStorage, and assertHasAttachmentFromStorageDisk.
```

As you might expect, the "HTML" assertions assert that the HTML version of your mailable contains a given string, while the "text" assertions assert that the plain-text version of your mailable contains a given string:

```

use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content(): void
{
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertFrom('jeffrey@example.com');
    $mailable->assertTo('taylor@example.com');
    $mailable->assertHasCc('abigail@example.com');
    $mailable->assertHasBcc('victoria@example.com');
    $mailable->assertHasReplyTo('tyler@example.com');
    $mailable->assertHasSubject('Invoice Paid');
    $mailable->assertHasTag('example-tag');
    $mailable->assertHasMetadata('key', 'value');

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');
    $mailable->assertSeeInOrderInHtml(['Invoice Paid', 'Thanks']);

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInOrderInText(['Invoice Paid', 'Thanks']);

    $mailable->assertHasAttachment('/path/to/file');
    $mailable->assertHasAttachment(Attachment::fromPath('/path/to/file'));
    $mailable->assertHasAttachedData($pdfData, 'name.pdf', ['mime' => 'application/pdf']);
    $mailable->assertHasAttachmentFromStorage('/path/to/file', 'name.pdf', ['mime' =>
    'application/pdf']);
    $mailable->assertHasAttachmentFromStorageDisk('s3', '/path/to/file', 'name.pdf', ['mime' =>
    'application/pdf']);
}

```

Testing Mailable Sending

We suggest testing the content of your mailables separately from your tests that assert that a given mailable was "sent" to a specific user. Typically, the content of mailables is not relevant to the code you are testing, and it is sufficient to simply assert that Laravel was instructed to send a given mailable.

You may use the `Mail` facade's `fake` method to prevent mail from being sent. After calling the `Mail` facade's `fake` method, you may then assert that mailables were instructed to be sent to users and even inspect the data the mailables received:

```
<?php

namespace Tests\Feature;

use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Mail::fake();

        // Perform order shipping...

        // Assert that no mailables were sent...
        Mail::assertNothingSent();

        // Assert that a mailable was sent...
        Mail::assertSent(OrderShipped::class);

        // Assert a mailable was sent twice...
        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);

        // Assert 3 total mailables were sent...
        Mail::assertSentCount(3);
    }
}
```

If you are queueing mailables for delivery in the background, you should use the `assertQueued` method instead of `assertSent`:

```
Mail::assertQueued(OrderShipped::class);
Mail::assertNotQueued(OrderShipped::class);
Mail::assertNothingQueued();
Mail::assertQueuedCount(3);
```

You may pass a closure to the `assertSent`, `assertNotSent`, `assertQueued`, or `assertNotQueued` methods in order to assert that a mailable was sent that passes a given "truth test". If at least one mailable was sent that passes the given truth test then the assertion will be successful:

```
Mail::assertSent(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

When calling the `Mail` facade's assertion methods, the mailable instance accepted by the provided closure exposes helpful methods for examining the mailable:

```
Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($user) {
    return $mail->hasTo($user->email) &&
        $mail->hasCc('...') &&
        $mail->hasBcc('...') &&
        $mail->hasReplyTo('...') &&
        $mail->hasFrom('...') &&
        $mail->hasSubject('...');
});
```

The mailable instance also includes several helpful methods for examining the attachments on a mailable:

```
use Illuminate\Mail\Mailables\Attachment;

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromPath('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromStorageDisk('s3', '/path/to/file')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($pdfData) {
    return $mail->hasAttachment(
        Attachment::fromData(fn () => $pdfData, 'name.pdf')
    );
});
```

You may have noticed that there are two methods for asserting that mail was not sent: `assertNotSent` and `assertNotQueued`. Sometimes you may wish to assert that no mail was sent **or** queued. To accomplish this, you may use the `assertNothingOutgoing` and `assertNotOutgoing` methods:

```
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

Mail and Local Development

When developing an application that sends email, you probably don't want to actually send emails to live email addresses. Laravel provides several ways to "disable" the actual sending of emails during local development.

Log Driver

Instead of sending your emails, the `log` mail driver will write all email messages to your log files for inspection. Typically, this driver would only be used during local development. For more information on configuring your application per environment, check out the [configuration documentation](#).

HELO / Mailtrap / Mailpit

Alternatively, you may use a service like [HELO](#) or [Mailtrap](#) and the `smtp` driver to send your email messages to a "dummy" mailbox where you may view them in a true email client. This approach has the benefit of allowing you to actually inspect the final emails in Mailtrap's message viewer.

If you are using [Laravel Sail](#), you may preview your messages using [Mailpit](#). When Sail is running, you may access the Mailpit interface at: <http://localhost:8025>.

Using a Global `to` Address

Finally, you may specify a global "to" address by invoking the `alwaysTo` method offered by the `Mail` facade. Typically, this method should be called from the `boot` method of one of your application's service providers:

```
use Illuminate\Support\Facades\Mail;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    if ($this->app->environment('local')) {
        Mail::alwaysTo('taylor@example.com');
    }
}
```

Events

Laravel fires two events during the process of sending mail messages. The `MessageSending` event is fired prior to a message being sent, while the `MessageSent` event is fired after a message has been sent. Remember, these events are fired when the mail is being `sent`, not when it is queued. You may register event listeners for this event in your `App\Providers\EventServiceProvider` service provider:

```
use App\Listeners\LogSendingMessage;
use App\Listeners\LogSentMessage;
use Illuminate\Mail\Events\MessageSending;
use Illuminate\Mail\Events\MessageSent;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    MessageSending::class => [
        LogSendingMessage::class,
    ],
    MessageSent::class => [
        LogSentMessage::class,
    ],
];
```

Custom Transports

Laravel includes a variety of mail transports; however, you may wish to write your own transports to deliver email via other services that Laravel does not support out of the box. To get started, define a class that extends the `Symfony\Component\Mailer\Transport\AbstractTransport` class. Then, implement the `doSend` and `__toString()` methods on your transport:

```
use MailchimpTransactional\ApiClient;
use Symfony\Component\Mailer\SentMessage;
use Symfony\Component\Mailer\Transport\AbstractTransport;
use Symfony\Component\Mime\Address;
use Symfony\Component\Mime\MessageConverter;

class MailchimpTransport extends AbstractTransport
{
    /**
     * Create a new Mailchimp transport instance.
     */
    public function __construct(
        protected ApiClient $client,
    ) {
        parent::__construct();
    }

    /**
     * {@inheritDoc}
     */
    protected function doSend(SentMessage $message): void
    {
        $email = MessageConverter::toEmail($message->getOriginalMessage());

        $this->client->messages->send(['message' => [
            'from_email' => $email->getFrom(),
            'to' => collect($email->getTo())->map(function (Address $email) {
                return ['email' => $email->getAddress(), 'type' => 'to'];
            })->all(),
            'subject' => $email->getSubject(),
            'text' => $email->getTextBody(),
        ]]);
    }

    /**
     * Get the string representation of the transport.
     */
    public function __toString(): string
    {
        return 'mailchimp';
    }
}
```

Once you've defined your custom transport, you may register it via the `extend` method provided by the `Mail` facade. Typically, this should be done within the `boot` method of your application's `AppServiceProvider` service provider. A `$config` argument will be passed to the closure provided to the `extend` method. This argument will contain the configuration array defined for the mailer in the application's `config/mail.php` configuration file:

```
use App\Mail\MailchimpTransport;
use Illuminate\Support\Facades\Mail;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Mail::extend('mailchimp', function (array $config = []) {
        return new MailchimpTransport(/* ... */);
    });
}
```

Once your custom transport has been defined and registered, you may create a mailer definition within your application's `config/mail.php` configuration file that utilizes the new transport:

```
'mailchimp' => [
    'transport' => 'mailchimp',
    // ...
],
```

Additional Symfony Transports

Laravel includes support for some existing Symfony maintained mail transports like Mailgun and Postmark. However, you may wish to extend Laravel with support for additional Symfony maintained transports. You can do so by requiring the necessary Symfony mailer via Composer and registering the transport with Laravel. For example, you may install and register the "Brevo" (formerly "Sendinblue") Symfony mailer:

```
composer require symfony/brevo-mailer symfony/http-client
```

Once the Brevo mailer package has been installed, you may add an entry for your Brevo API credentials to your application's `services` configuration file:

```
'brevo' => [
    'key' => 'your-api-key',
],
```

Next, you may use the `Mail` facade's `extend` method to register the transport with Laravel. Typically, this should be done within the `boot` method of a service provider:

```
use Illuminate\Support\Facades\Mail;
use Symfony\Component\Mailer\Bridge\Brevo\Transport\BrevoTransportFactory;
use Symfony\Component\Mailer\Transport\Dsn;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Mail::extend('brevo', function () {
        return (new BrevoTransportFactory)->create(
            new Dsn(
                'brevo+api',
                'default',
                config('services.brevo.key')
            )
        );
    });
}
```

Once your transport has been registered, you may create a mailer definition within your application's config/mail.php configuration file that utilizes the new transport:

```
'brevo' => [
    'transport' => 'brevo',
    // ...
],
```

Notifications

- [Introduction](#)
- [Generating Notifications](#)
- [Sending Notifications](#)
 - [Using the Notifiable Trait](#)
 - [Using the Notification Facade](#)
 - [Specifying Delivery Channels](#)
 - [Queueing Notifications](#)
 - [On-Demand Notifications](#)
- [Mail Notifications](#)
 - [Formatting Mail Messages](#)
 - [Customizing the Sender](#)
 - [Customizing the Recipient](#)
 - [Customizing the Subject](#)
 - [Customizing the Mailer](#)
 - [Customizing the Templates](#)
 - [Attachments](#)
 - [Adding Tags and Metadata](#)
 - [Customizing the Symfony Message](#)
 - [Using Mailables](#)
 - [Previewing Mail Notifications](#)
- [Markdown Mail Notifications](#)
 - [Generating the Message](#)
 - [Writing the Message](#)
 - [Customizing the Components](#)
- [Database Notifications](#)
 - [Prerequisites](#)
 - [Formatting Database Notifications](#)
 - [Accessing the Notifications](#)
 - [Marking Notifications as Read](#)
- [Broadcast Notifications](#)
 - [Prerequisites](#)
 - [Formatting Broadcast Notifications](#)
 - [Listening for Notifications](#)
- [SMS Notifications](#)
 - [Prerequisites](#)
 - [Formatting SMS Notifications](#)
 - [Unicode Content](#)
 - [Customizing the "From" Number](#)
 - [Adding a Client Reference](#)
 - [Routing SMS Notifications](#)
- [Slack Notifications](#)
 - [Prerequisites](#)
 - [Formatting Slack Notifications](#)
 - [Slack Interactivity](#)
 - [Routing Slack Notifications](#)
 - [Notifying External Slack Workspaces](#)
- [Localizing Notifications](#)
- [Testing](#)
- [Notification Events](#)

- [Custom Channels](#)

Introduction

In addition to support for [sending email](#), Laravel provides support for sending notifications across a variety of delivery channels, including email, SMS (via [Vonage](#), formerly known as Nexmo), and [Slack](#). In addition, a variety of [community built notification channels](#) have been created to send notifications over dozens of different channels! Notifications may also be stored in a database so they may be displayed in your web interface.

Typically, notifications should be short, informational messages that notify users of something that occurred in your application. For example, if you are writing a billing application, you might send an "Invoice Paid" notification to your users via the email and SMS channels.

Generating Notifications

In Laravel, each notification is represented by a single class that is typically stored in the `app/Notifications` directory. Don't worry if you don't see this directory in your application - it will be created for you when you run the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid
```

This command will place a fresh notification class in your `app/Notifications` directory. Each notification class contains a `via` method and a variable number of message building methods, such as `toMail` or `toDatabase`, that convert the notification to a message tailored for that particular channel.

Sending Notifications

Using the Notifiable Trait

Notifications may be sent in two ways: using the `notify` method of the `Notifiable` trait or using the `Notification facade`. The `Notifiable` trait is included on your application's `App\Models\User` model by default:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

The `notify` method that is provided by this trait expects to receive a notification instance:

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```



Remember, you may use the `Notifiable` trait on any of your models. You are not limited to only including it on your `User` model.

Using the Notification Facade

Alternatively, you may send notifications via the `Notification` facade. This approach is useful when you need to send a notification to multiple notifiable entities such as a collection of users. To send notifications using the facade, pass all of the notifiable entities and the notification instance to the `send` method:

```
use Illuminate\Support\Facades\Notification;

Notification::send($users, new InvoicePaid($invoice));
```

You can also send notifications immediately using the `sendNow` method. This method will send the notification immediately even if the notification implements the `ShouldQueue` interface:

```
Notification::sendNow($developers, new DeploymentCompleted($deployment));
```

Specifying Delivery Channels

Every notification class has a `via` method that determines on which channels the notification will be delivered. Notifications may be sent on the `mail`, `database`, `broadcast`, `vonage`, and `slack` channels.



If you would like to use other delivery channels such as Telegram or Pusher, check out the community driven [Laravel Notification Channels website](#).

The `via` method receives a `$notifiable` instance, which will be an instance of the class to which the notification is being sent. You may use `$notifiable` to determine which channels the notification should be delivered on:

```
/**
 * Get the notification's delivery channels.
 *
 * @return array<int, string>
 */
public function via(object $notifiable): array
{
    return $notifiable->prefers_sms ? ['vonage'] : ['mail', 'database'];
}
```

Queueing Notifications



Before queueing notifications you should configure your queue and [start a worker](#).

Sending notifications can take time, especially if the channel needs to make an external API call to deliver the notification. To speed up your application's response time, let your notification be queued by adding the `ShouldQueue` interface and `Queueable` trait to your class. The interface and trait are already imported for all notifications generated using the `make:notification` command, so you may immediately add them to your notification class:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

Once the `ShouldQueue` interface has been added to your notification, you may send the notification like normal. Laravel will detect the `ShouldQueue` interface on the class and automatically queue the delivery of the notification:

```
$user->notify(new InvoicePaid($invoice));
```

When queueing notifications, a queued job will be created for each recipient and channel combination. For example, six jobs will be dispatched to the queue if your notification has three recipients and two channels.

Delaying Notifications

If you would like to delay the delivery of the notification, you may chain the `delay` method onto your notification instantiation:

```
$delay = now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($delay));
```

Delaying Notifications per Channel

You may pass an array to the `delay` method to specify the delay amount for specific channels:

```
$user->notify((new InvoicePaid($invoice))->delay([
    'mail' => now()->addMinutes(5),
    'sms' => now()->addMinutes(10),
]));
```

Alternatively, you may define a `withDelay` method on the notification class itself. The `withDelay` method should return an array of channel names and delay values:

```
/**
 * Determine the notification's delivery delay.
 *
 * @return array<string, \Illuminate\Support\Carbon>
 */
public function withDelay(object $notifiable): array
{
    return [
        'mail' => now()>addMinutes(5),
        'sms' => now()>addMinutes(10),
    ];
}
```

Customizing the Notification Queue Connection

By default, queued notifications will be queued using your application's default queue connection. If you would like to specify a different connection that should be used for a particular notification, you may call the `onConnection` method from your notification's constructor:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * Create a new notification instance.
     */
    public function __construct()
    {
        $this->onConnection('redis');
    }
}
```

Or, if you would like to specify a specific queue connection that should be used for each notification channel supported by the notification, you may define a `viaConnections` method on your notification. This method should return an array of channel name / queue connection name pairs:

```
/**
 * Determine which connections should be used for each notification channel.
 *
 * @return array<string, string>
 */
public function viaConnections(): array
{
    return [
        'mail' => 'redis',
        'database' => 'sync',
    ];
}
```

Customizing Notification Channel Queues

If you would like to specify a specific queue that should be used for each notification channel supported by the notification, you may define a `viaQueues` method on your notification. This method should return an array of channel name / queue name pairs:

```
/**
 * Determine which queues should be used for each notification channel.
 *
 * @return array<string, string>
 */
public function viaQueues(): array
{
    return [
        'mail' => 'mail-queue',
        'slack' => 'slack-queue',
    ];
}
```

Queued Notifications and Database Transactions

When queued notifications are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your notification depends on these models, unexpected errors can occur when the job that sends the queued notification is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued notification should be dispatched after all open database transactions have been committed by calling the `afterCommit` method when sending the notification:

```
use App\Notifications\InvoicePaid;

$user->notify((new InvoicePaid($invoice))->afterCommit());
```

Alternatively, you may call the `afterCommit` method from your notification's constructor:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * Create a new notification instance.
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```



To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).

Determining if a Queued Notification Should Be Sent

After a queued notification has been dispatched for the queue for background processing, it will typically be accepted by a queue worker and sent to its intended recipient.

However, if you would like to make the final determination on whether the queued notification should be sent after it is being processed by a queue worker, you may define a `shouldSend` method on the notification class. If this method returns `false`, the notification will not be sent:

```
/**
 * Determine if the notification should be sent.
 */
public function shouldSend(object $notifiable, string $channel): bool
{
    return $this->invoice->isPaid();
}
```

On-Demand Notifications

Sometimes you may need to send a notification to someone who is not stored as a "user" of your application. Using the `Notification` facade's `route` method, you may specify ad-hoc notification routing information before sending the notification:

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Support\Facades\Notification;

Notification::route('mail', 'taylor@example.com')
    ->route('vonage', '5555555555')
    ->route('slack', '#slack-channel')
    ->route('broadcast', [new Channel('channel-name')])
    ->notify(new InvoicePaid($invoice));
```

If you would like to provide the recipient's name when sending an on-demand notification to the `mail` route, you may provide an array that contains the email address as the key and the name as the value of the first element in the array:

```
Notification::route('mail', [
    'barrett@example.com' => 'Barrett Blair',
])->notify(new InvoicePaid($invoice));
```

Using the `routes` method, you may provide ad-hoc routing information for multiple notification channels at once:

```
Notification::routes([
    'mail' => ['barrett@example.com' => 'Barrett Blair'],
    'vonage' => '5555555555',
])->notify(new InvoicePaid($invoice));
```

Mail Notifications

Formatting Mail Messages

If a notification supports being sent as an email, you should define a `toMail` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\MailMessage` instance.

The `MailMessage` class contains a few simple methods to help you build transactional email messages. Mail messages may contain lines of text as well as a "call to action". Let's take a look at an example `toMail` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->greeting('Hello!')
        ->line('One of your invoices has been paid!')
        ->lineIf($this->amount > 0, "Amount paid: {$this->amount}")
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```



Note we are using `$this->invoice->id` in our `toMail` method. You may pass any data your notification needs to generate its message into the notification's constructor.

In this example, we register a greeting, a line of text, a call to action, and then another line of text. These methods provided by the `MailMessage` object make it simple and fast to format small transactional emails. The mail channel will then translate the message components into a beautiful, responsive HTML email template with a plain-text counterpart. Here is an example of an email generated by the `mail` channel:

The screenshot shows a mail client interface with a dark sidebar on the left containing icons for reply, forward, share, and settings. The main window title is "HELO". The message subject is "Invoice Paid". It was sent at 10/22/2020, 12:54 PM, has a size of 12.99 kB, and contains "No broken links". Action buttons include "Delete", "Share", and "Forward". The message details show "From: Laravel <hello@laravel.com>", "To: taylor@laravel.com", and "Message ID: <cc3d3e53eea76c69c40ff7339bb35e1b@swift.generated>". Below these are tabs for "HTML", "HTML-Source", "Text", "Raw", "Debug", "Headers", "Links", "Spam Report (0)", and "Testing". The main content area displays an invoice notification template with a Laravel logo, a "Hello!" greeting, a message stating "One of your invoices has been paid!", a "View Invoice" button, a "Thank you for using our application!" message, and a "Regards, Laravel" signature. A note at the bottom says, "If you're having trouble clicking the 'View Invoice' button, copy and paste the URL below into your web browser: <http://laravel.test/invoice/1>". The footer of the message area includes the copyright notice "© 2020 Laravel. All rights reserved."



When sending mail notifications, be sure to set the `name` configuration option in your `config/app.php` configuration file. This value will be used in the header and footer of your mail notification messages.

Error Messages

Some notifications inform users of errors, such as a failed invoice payment. You may indicate that a mail message is regarding an error by calling the `error` method when building your message. When using the `error` method on a mail message, the call to action button will be red instead of black:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->error()
        ->subject('Invoice Payment Failed')
        ->line('...');

}
```

Other Mail Notification Formatting Options

Instead of defining the "lines" of text in the notification class, you may use the `view` method to specify a custom template that should be used to render the notification email:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        'mail.invoice.paid', ['invoice' => $this->invoice]
    );
}
```

You may specify a plain-text view for the mail message by passing the view name as the second element of an array that is given to the `view` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        ['mail.invoice.paid', 'mail.invoice.paid-text'],
        ['invoice' => $this->invoice]
    );
}
```

Or, if your message only has a plain-text view, you may utilize the `text` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->text(
        'mail.invoice.paid-text', ['invoice' => $this->invoice]
    );
}
```

Customizing the Sender

By default, the email's sender / from address is defined in the `config/mail.php` configuration file. However, you may specify the from address for a specific notification using the `from` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->from('barrett@example.com', 'Barrett Blair')
        ->line('...');
}
```

Customizing the Recipient

When sending notifications via the `mail` channel, the notification system will automatically look for an `email` property on your notifiable entity. You may customize which email address is used to deliver the notification by defining a `routeNotificationForMail` method on the notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @return array<string, string>|string
     */
    public function routeNotificationForMail(Notification $notification): array|string
    {
        // Return email address only...
        return $this->email_address;

        // Return email address and name...
        return [$this->email_address => $this->name];
    }
}
```

Customizing the Subject

By default, the email's subject is the class name of the notification formatted to "Title Case". So, if your notification class is named `[InvoicePaid]`, the email's subject will be `[Invoice Paid]`. If you would like to specify a different subject for the message, you may call the `subject` method when building your message:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->subject('Notification Subject')
        ->line('...');
}
```

Customizing the Mailer

By default, the email notification will be sent using the default mailer defined in the `[config/mail.php]` configuration file. However, you may specify a different mailer at runtime by calling the `mailer` method when building your message:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->mailer('postmark')
        ->line('...');
}
```

Customizing the Templates

You can modify the HTML and plain-text template used by mail notifications by publishing the notification package's resources. After running this command, the mail notification templates will be located in the `resources/views/vendor/notifications` directory:

```
php artisan vendor:publish --tag=laravel-notifications
```

Attachments

To add attachments to an email notification, use the `attach` method while building your message. The `attach` method accepts the absolute path to the file as its first argument:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->greeting('Hello!')
        ->attach('/path/to/file');
}
```



The `attach` method offered by notification mail messages also accepts [attachable objects](#). Please consult the comprehensive [attachable object documentation](#) to learn more.

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->greeting('Hello!')
        ->attach('/path/to/file', [
            'as' => 'name.pdf',
            'mime' => 'application/pdf',
        ]);
}
```

Unlike attaching files in mailable objects, you may not attach a file directly from a storage disk using `attachFromStorage`. You should rather use the `attach` method with an absolute path to the file on the storage disk. Alternatively, you could return a `mailable` from the `toMail` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): Mailable
{
    return (new InvoicePaidMailable($this->invoice))
        ->to($notifiable->email)
        ->attachFromStorage('/path/to/file');
}
```

When necessary, multiple files may be attached to a message using the `attachMany` method:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->greeting('Hello!')
        ->attachMany([
            '/path/to/forge.svg',
            '/path/to/vapor.svg' => [
                'as' => 'Logo.svg',
                'mime' => 'image/svg+xml',
            ],
        ]);
}
```

Raw Data Attachments

The `attachData` method may be used to attach a raw string of bytes as an attachment. When calling the `attachData` method, you should provide the filename that should be assigned to the attachment:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->greeting('Hello!')
        ->attachData($this->pdf, 'name.pdf', [
            'mime' => 'application/pdf',
        ]);
}
```

Adding Tags and Metadata

Some third-party email providers such as Mailgun and Postmark support message "tags" and "metadata", which may be used to group and track emails sent by your application. You may add tags and metadata to an email message via the `tag` and `metadata` methods:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->greeting('Comment Upvoted!')
        ->tag('upvote')
        ->metadata('comment_id', $this->comment->id);
}
```

If your application is using the Mailgun driver, you may consult Mailgun's documentation for more information on [tags](#) and [metadata](#). Likewise, the Postmark documentation may also be consulted for more information on their support for [tags](#) and [metadata](#).

If your application is using Amazon SES to send emails, you should use the `metadata` method to attach [SES "tags"](#) to the message.

Customizing the Symfony Message

The `withSymfonyMessage` method of the `MailMessage` class allows you to register a closure which will be invoked with the Symfony Message instance before sending the message. This gives you an opportunity to deeply customize the message before it is delivered:

```
use Symfony\Component\Mime\Email;

/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->withSymfonyMessage(function (Email $message) {
            $message->getHeaders()->addTextHeader(
                'Custom-Header', 'Header Value'
            );
        });
}
```

Using Mailables

If needed, you may return a full `mailable` object from your notification's `toMail` method. When returning a `Mailable` instead of a `MailMessage`, you will need to specify the message recipient using the mailable object's `to` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Mail\Mailable;

/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): Mailable
{
    return (new InvoicePaidMailable($this->invoice))
        ->to($notifiable->email);
}
```

Mailables and On-Demand Notifications

If you are sending an `on-demand notification`, the `$notifiable` instance given to the `toMail` method will be an instance of `Illuminate\Notifications\AnonymousNotifiable`, which offers a `routeNotificationFor` method that may be used to retrieve the email address the on-demand notification should be sent to:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Notifications\AnonymousNotifiable;
use Illuminate\Mail\Mailable;

/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): Mailable
{
    $address = $notifiable instanceof AnonymousNotifiable
        ? $notifiable->routeNotificationFor('mail')
        : $notifiable->email;

    return (new InvoicePaidMailable($this->invoice))
        ->to($address);
}
```

Previewing Mail Notifications

When designing a mail notification template, it is convenient to quickly preview the rendered mail message in your browser like a typical Blade template. For this reason, Laravel allows you to return any mail message generated by a mail notification directly from a route closure or controller. When a `MailMessage` is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
use App\Models\Invoice;
use App\Notifications\InvoicePaid;

Route::get('/notification', function () {
    $invoice = Invoice::find(1);

    return (new InvoicePaid($invoice))
        ->toMail($invoice->user);
});
```

Markdown Mail Notifications

Markdown mail notifications allow you to take advantage of the pre-built templates of mail notifications, while giving you more freedom to write longer, customized messages. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating the Message

To generate a notification with a corresponding Markdown template, you may use the `--markdown` option of the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

Like all other mail notifications, notifications that use Markdown templates should define a `toMail` method on their notification class. However, instead of using the `line` and `action` methods to construct the notification, use the `markdown` method to specify the name of the Markdown template that should be used. An array of data you wish to make available to the template may be passed as the method's second argument:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Writing the Message

Markdown mail notifications use a combination of Blade components and Markdown syntax which allow you to easily construct notifications while leveraging Laravel's pre-crafted notification components:

```
<x-mail::message>
# Invoice Paid

Your invoice has been paid!

<x-mail::button :url="$url">
View Invoice
</x-mail::button>

Thanks,<br>
{{ config('app.name') }}
</x-mail::message>
```

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `green`, and `red`. You may add as many button components to a notification as you wish:

```
<x-mail::button :url="$url" color="green">
View Invoice
</x-mail::button>
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the notification. This allows you to draw attention to a given block of text:

```
<x-mail::panel>
This is the panel content.
</x-mail::panel>
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
<x-mail::table>
| Laravel      | Table       | Example   |
| ----- |:-----:| -----:|
| Col 2 is    | Centered   | $10        |
| Col 3 is    | Right-Aligned | $20        |
</x-mail::table>
```

Customizing the Components

You may export all of the Markdown notification components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

```
php artisan vendor:publish --tag=laravel-mail
```

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing the CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be in-lined within the HTML representations of your Markdown notifications.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of the `mail` configuration file to match the name of your new theme.

To customize the theme for an individual notification, you may call the `theme` method while building the notification's mail message. The `theme` method accepts the name of the theme that should be used when sending the notification:

```
/**
 * Get the mail representation of the notification.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->theme('invoice')
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Database Notifications

Prerequisites

The `database` notification channel stores the notification information in a database table. This table will contain information such as the notification type as well as a JSON data structure that describes the notification.

You can query the table to display the notifications in your application's user interface. But, before you can do that, you will need to create a database table to hold your notifications. You may use the `notifications:table` command to generate a [migration](#) with the proper table schema:

```
php artisan notifications:table

php artisan migrate
```



If your notifiable models are using [UUID](#) or [ULID](#) primary keys, you should replace the `morphs` method with `uuidMorphs` or `ulidMorphs` in the notification table migration.

Formatting Database Notifications

If a notification supports being stored in a database table, you should define a `toDatabase` or `toArray` method on the notification class. This method will receive a `$notifiable` entity and should return a plain PHP array. The returned array will be encoded as JSON and stored in the `data` column of your `notifications` table. Let's take a look at an example `toArray` method:

```
/**
 * Get the array representation of the notification.
 *
 * @return array<string, mixed>
 */
public function toArray(object $notifiable): array
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

When the notification is stored in your application's database, the `type` column will be populated with the notification's class name. However, you may customize this behavior by defining a `databaseType` method on your notification class:

```
/**
 * Get the notification's database type.
 *
 * @return string
 */
public function databaseType(object $notifiable): string
{
    return 'invoice-paid';
}
```

`toDatabase` vs. `toArray`

The `toArray` method is also used by the `broadcast` channel to determine which data to broadcast to your JavaScript powered frontend. If you would like to have two different array representations for the `database` and `broadcast` channels, you should define a `toDatabase` method instead of a `toArray` method.

Accessing the Notifications

Once notifications are stored in the database, you need a convenient way to access them from your notifiable entities. The `Illuminate\Notifications\Notifiable` trait, which is included on Laravel's default `App\Models\User` model, includes a `notifications` [Eloquent relationship](#) that returns the notifications for the entity. To fetch notifications, you may access this method like any other Eloquent relationship. By default, notifications will be sorted by the `created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

If you want to retrieve only the "unread" notifications, you may use the `unreadNotifications` relationship. Again, these notifications will be sorted by the `created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```



To access your notifications from your JavaScript client, you should define a notification controller for your application which returns the notifications for a notifiable entity, such as the current user. You may then make an HTTP request to that controller's URL from your JavaScript client.

Marking Notifications as Read

Typically, you will want to mark a notification as "read" when a user views it. The `Illuminate\Notifications\Notifiable` trait provides a `markAsRead` method, which updates the `read_at` column on the notification's database record:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

However, instead of looping through each notification, you may use the `markAsRead` method directly on a collection of notifications:

```
$user->unreadNotifications->markAsRead();
```

You may also use a mass-update query to mark all of the notifications as read without retrieving them from the database:

```
$user = App\Models\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

You may `delete` the notifications to remove them from the table entirely:

```
$user->notifications()->delete();
```

Broadcast Notifications

Prerequisites

Before broadcasting notifications, you should configure and be familiar with Laravel's [event broadcasting](#) services. Event broadcasting provides a way to react to server-side Laravel events from your JavaScript powered frontend.

Formatting Broadcast Notifications

The `broadcast` channel broadcasts notifications using Laravel's [event broadcasting](#) services, allowing your JavaScript powered frontend to catch notifications in realtime. If a notification supports broadcasting, you can define a `toBroadcast` method on the notification class. This method will receive a `$notifiable` entity and should return a `BroadcastMessage` instance. If the `toBroadcast` method does not exist, the `toArray` method will be used to gather

the data that should be broadcast. The returned data will be encoded as JSON and broadcast to your JavaScript powered frontend. Let's take a look at an example `toBroadcast` method:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the broadcastable representation of the notification.
 */
public function toBroadcast(object $notifiable): BroadcastMessage
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

Broadcast Queue Configuration

All broadcast notifications are queued for broadcasting. If you would like to configure the queue connection or queue name that is used to queue the broadcast operation, you may use the `onConnection` and `onQueue` methods of the `BroadcastMessage`:

```
return (new BroadcastMessage($data))
    ->onConnection('sq')
    ->onQueue('broadcasts');
```

Customizing the Notification Type

In addition to the data you specify, all broadcast notifications also have a `type` field containing the full class name of the notification. If you would like to customize the notification `type`, you may define a `broadcastType` method on the notification class:

```
/**
 * Get the type of the notification being broadcast.
 */
public function broadcastType(): string
{
    return 'broadcast.message';
}
```

Listening for Notifications

Notifications will broadcast on a private channel formatted using a `{notifiable}.{id}` convention. So, if you are sending a notification to an `App\Models\User` instance with an ID of `1`, the notification will be broadcast on the `App.Models.User.1` private channel. When using [Laravel Echo](#), you may easily listen for notifications on a channel using the `notification` method:

```
Echo.private('App.Models.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
});
```

Customizing the Notification Channel

If you would like to customize which channel that an entity's broadcast notifications are broadcast on, you may define a `receivesBroadcastNotificationsOn` method on the notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The channels the user receives notification broadcasts on.
     */
    public function receivesBroadcastNotificationsOn(): string
    {
        return 'users.'.$this->id;
    }
}
```

SMS Notifications

Prerequisites

Sending SMS notifications in Laravel is powered by [Vonage](#) (formerly known as Nexmo). Before you can send notifications via Vonage, you need to install the `laravel/vonage-notification-channel` and `guzzlehttp/guzzle` packages:

```
composer require laravel/vonage-notification-channel guzzlehttp/guzzle
```

The package includes a [configuration file](#). However, you are not required to export this configuration file to your own application. You can simply use the `VONAGE_KEY` and `VONAGE_SECRET` environment variables to define your Vonage public and secret keys.

After defining your keys, you should set a `VONAGE_SMS_FROM` environment variable that defines the phone number that your SMS messages should be sent from by default. You may generate this phone number within the Vonage control panel:

```
VONAGE_SMS_FROM=15556666666
```

Formatting SMS Notifications

If a notification supports being sent as an SMS, you should define a `toVonage` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\VonageMessage` instance:

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * Get the Vonage / SMS representation of the notification.
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->content('Your SMS message content');
}
```

Unicode Content

If your SMS message will contain unicode characters, you should call the `unicode` method when constructing the `VonageMessage` instance:

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * Get the Vonage / SMS representation of the notification.
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->content('Your unicode message')
        ->unicode();
}
```

Customizing the "From" Number

If you would like to send some notifications from a phone number that is different from the phone number specified by your `VONAGE_SMS_FROM` environment variable, you may call the `from` method on a `VonageMessage` instance:

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * Get the Vonage / SMS representation of the notification.
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->content('Your SMS message content')
        ->from('15554443333');
}
```

Adding a Client Reference

If you would like to keep track of costs per user, team, or client, you may add a "client reference" to the notification. Vonage will allow you to generate reports using this client reference so that you can better understand a particular customer's SMS usage. The client reference can be any string up to 40 characters:

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * Get the Vonage / SMS representation of the notification.
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->clientReference((string) $notifiable->id)
        ->content('Your SMS message content');
}
```

Routing SMS Notifications

To route Vonage notifications to the proper phone number, define a `routeNotificationForVonage` method on your notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Vonage channel.
     */
    public function routeNotificationForVonage(Notification $notification): string
    {
        return $this->phone_number;
    }
}
```

Slack Notifications

Prerequisites

Before sending Slack notifications, you should install the Slack notification channel via Composer:

```
composer require laravel/slack-notification-channel
```

Additionally, you must create a [Slack App](#) for your Slack workspace.

If you only need to send notifications to the same Slack workspace that the App is created in, you should ensure that your App has the `chat:write`, `chat:write.public`, and `chat:write.customize` scopes. These scopes can be added from the "OAuth & Permissions" App management tab within Slack.

Next, copy the App's "Bot User OAuth Token" and place it within a `slack` configuration array in your application's `services.php` configuration file. This token can be found on the "OAuth & Permissions" tab within Slack:

```
'slack' => [
    'notifications' => [
        'bot_user_oauth_token' => env('SLACK_BOT_USER_OAUTH_TOKEN'),
        'channel' => env('SLACK_BOT_USER_DEFAULT_CHANNEL'),
    ],
],
```

App Distribution

If your application will be sending notifications to external Slack workspaces that are owned by your application's users, you will need to "distribute" your App via Slack. App distribution can be managed from your App's "Manage Distribution" tab within Slack. Once your App has been distributed, you may use [Socialite](#) to [obtain Slack Bot tokens](#) on behalf of your application's users.

Formatting Slack Notifications

If a notification supports being sent as a Slack message, you should define a `toSlack` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Slack\SlackMessage` instance. You can construct rich notifications using [Slack's Block Kit API](#). The following example may be previewed in [Slack's Block Kit builder](#):

```
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;
use Illuminate\Notifications\Slack\BlockKit\Composites\ConfirmObject;
use Illuminate\Notifications\Slack\SlackMessage;

/**
 * Get the Slack representation of the notification.
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
        ->text('One of your invoices has been paid!')
        ->headerBlock('Invoice Paid')
        ->contextBlock(function (ContextBlock $block) {
            $block->text('Customer #1234');
        })
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('An invoice has been paid.');
            $block->field("*Invoice No:*\\n1000")->markdown();
            $block->field("*Invoice Recipient:*\\ntaylor@laravel.com")->markdown();
        })
        ->dividerBlock()
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('Congratulations!');
        });
}
```

Slack Interactivity

Slack's Block Kit notification system provides powerful features to [handle user interaction](#). To utilize these features, your Slack App should have "Interactivity" enabled and a "Request URL" configured that points to a URL served by your application. These settings can be managed from the "Interactivity & Shortcuts" App management tab within Slack.

In the following example, which utilizes the `actionsBlock` method, Slack will send a `POST` request to your "Request URL" with a payload containing the Slack user who clicked the button, the ID of the clicked button, and more. Your

application can then determine the action to take based on the payload. You should also [verify the request](#) was made by Slack:

```
use Illuminate\Notifications\Slack\BlockKit\Blocks\ActionsBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;
use Illuminate\Notifications\Slack\SlackMessage;

/**
 * Get the Slack representation of the notification.
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
        ->text('One of your invoices has been paid!')
        ->headerBlock('Invoice Paid')
        ->contextBlock(function (ContextBlock $block) {
            $block->text('Customer #1234');
        })
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('An invoice has been paid.');
        })
        ->actionsBlock(function (ActionsBlock $block) {
            // ID defaults to "button_acknowledge_invoice"...
            $block->button('Acknowledge Invoice')->primary();

            // Manually configure the ID...
            $block->button('Deny')->danger()->id('deny_invoice');
        });
}
```

Confirmation Modals

If you would like users to be required to confirm an action before it is performed, you may invoke the `confirm` method when defining your button. The `confirm` method accepts a message and a closure which receives a `ConfirmObject` instance:

```

use Illuminate\Notifications\Slack\BlockKit\Blocks\ActionsBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;
use Illuminate\Notifications\Slack\BlockKit\Composites\ConfirmObject;
use Illuminate\Notifications\Slack\SlackMessage;

/**
 * Get the Slack representation of the notification.
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
        ->text('One of your invoices has been paid!')
        ->headerBlock('Invoice Paid')
        ->contextBlock(function (ContextBlock $block) {
            $block->text('Customer #1234');
        })
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('An invoice has been paid.');
        })
        ->actionsBlock(function (ActionsBlock $block) {
            $block->button('Acknowledge Invoice')
                ->primary()
                ->confirm(
                    'Acknowledge the payment and send a thank you email?',
                    function (ConfirmObject $dialog) {
                        $dialog->confirm('Yes');
                        $dialog->deny('No');
                    }
                );
        });
}

```

Inspecting Slack Blocks

If you would like to quickly inspect the blocks you've been building, you can invoke the `dd` method on the `SlackMessage` instance. The `dd` method will generate and dump a URL to Slack's [Block Kit Builder](#), which displays a preview of the payload and notification in your browser. You may pass `true` to the `dd` method to dump the raw payload:

```

return (new SlackMessage)
    ->text('One of your invoices has been paid!')
    ->headerBlock('Invoice Paid')
    ->dd();

```

Routing Slack Notifications

To direct Slack notifications to the appropriate Slack team and channel, define a `routeNotificationForSlack` method on your notifiable model. This method can return one of three values:

- `null` - which defers routing to the channel configured in the notification itself. You may use the `to` method when building your `SlackMessage` to configure the channel within the notification.
- A string specifying the Slack channel to send the notification to, e.g. `#support-channel`.
- A `SlackRoute` instance, which allows you to specify an OAuth token and channel name, e.g. `SlackRoute::make($this->slack_channel, $this->slack_token)`. This method should be used to send notifications to external workspaces.

For instance, returning `#support-channel` from the `routeNotificationForSlack` method will send the notification to the `#support-channel` channel in the workspace associated with the Bot User OAuth token located in your application's `services.php` configuration file:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     */
    public function routeNotificationForSlack(Notification $notification): mixed
    {
        return '#support-channel';
    }
}
```

Notifying External Slack Workspaces



Before sending notifications to external Slack workspaces, your Slack App must be [distributed](#).

Of course, you will often want to send notifications to the Slack workspaces owned by your application's users. To do so, you will first need to obtain a Slack OAuth token for the user. Thankfully, [Laravel Socialite](#) includes a Slack driver that will allow you to easily authenticate your application's users with Slack and [obtain a bot token](#).

Once you have obtained the bot token and stored it within your application's database, you may utilize the `SlackRoute::make` method to route a notification to the user's workspace. In addition, your application will likely need to offer an opportunity for the user to specify which channel notifications should be sent to:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Slack\SlackRoute;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     */
    public function routeNotificationForSlack(Notification $notification): mixed
    {
        return SlackRoute::make($this->slack_channel, $this->slack_token);
    }
}
```

Localizing Notifications

Laravel allows you to send notifications in a locale other than the HTTP request's current locale, and will even remember this locale if the notification is queued.

To accomplish this, the `Illuminate\Notifications\Notification` class offers a `locale` method to set the desired language. The application will change into this locale when the notification is being evaluated and then revert back to the previous locale when evaluation is complete:

```
$user->notify((new InvoicePaid($invoice))->locale('es'));
```

Localization of multiple notifiable entries may also be achieved via the `Notification` facade:

```
Notification::locale('es')->send(
    $users, new InvoicePaid($invoice)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on your notifiable model, you may instruct Laravel to use this stored locale when sending a notification:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending notifications and mailables to the model. Therefore, there is no need to call the `Locale` method when using this interface:

```
$user->notify(new InvoicePaid($invoice));
```

Testing

You may use the `Notification` facade's `fake` method to prevent notifications from being sent. Typically, sending notifications is unrelated to the code you are actually testing. Most likely, it is sufficient to simply assert that Laravel was instructed to send a given notification.

After calling the `Notification` facade's `fake` method, you may then assert that notifications were instructed to be sent to users and even inspect the data the notifications received:

```
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Notification::fake();

        // Perform order shipping...

        // Assert that no notifications were sent...
        Notification::assertNothingSent();

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );

        // Assert that a given number of notifications were sent...
        Notification::assertCount(3);
    }
}
```

You may pass a closure to the `assertSentTo` or `assertNotSentTo` methods in order to assert that a notification was sent that passes a given "truth test". If at least one notification was sent that passes the given truth test then the assertion will be successful:

```
Notification::assertSentTo(
    $user,
    function (OrderShipped $notification, array $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);
```

On-Demand Notifications

If the code you are testing sends [on-demand notifications](#), you can test that the on-demand notification was sent via the `assertSentOnDemand` method:

```
Notification::assertSentOnDemand(OrderShipped::class);
```

By passing a closure as the second argument to the `assertSentOnDemand` method, you may determine if an on-demand notification was sent to the correct "route" address:

```
Notification::assertSentOnDemand(
    OrderShipped::class,
    function (OrderShipped $notification, array $channels, object $notifiable) use ($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
```

Notification Events

Notification Sending Event

When a notification is sending, the `Illuminate\Notifications\Events\NotificationSending` event is dispatched by the notification system. This contains the "notifiable" entity and the notification instance itself. You may register listeners for this event in your application's `EventServiceProvider`:

```
use App\Listeners\CheckNotificationStatus;
use Illuminate\Notifications\Events\NotificationSending;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    NotificationSending::class => [
        CheckNotificationStatus::class,
    ],
];
```

The notification will not be sent if an event listener for the `NotificationSending` event returns `false` from its `handle` method:

```
use Illuminate\Notifications\Events\NotificationSending;

/**
 * Handle the event.
 */
public function handle(NotificationSending $event): bool
{
    return false;
}
```

Within an event listener, you may access the `notifiable`, `notification`, and `channel` properties on the event to learn more about the notification recipient or the notification itself:

```
/**
 * Handle the event.
 */
public function handle(NotificationSending $event): void
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

Notification Sent Event

When a notification is sent, the `Illuminate\Notifications\Events\NotificationSent` event is dispatched by the notification system. This contains the "notifiable" entity and the notification instance itself. You may register listeners for this event in your `EventServiceProvider`:

```
use App\Listeners\LogNotification;
use Illuminate\Notifications\Events\NotificationSent;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    NotificationSent::class => [
        LogNotification::class,
    ],
];
```



After registering listeners in your `EventServiceProvider`, use the `event:generate` Artisan command to quickly generate listener classes.

Within an event listener, you may access the `notifiable`, `notification`, `channel`, and `response` properties on the event to learn more about the notification recipient or the notification itself:

```
/**
 * Handle the event.
 */
public function handle(NotificationSent $event): void
{
    // $event->channel
    // $event->notifiable
    // $event->notification
    // $event->response
}
```

Custom Channels

Laravel ships with a handful of notification channels, but you may want to write your own drivers to deliver notifications via other channels. Laravel makes it simple. To get started, define a class that contains a `send` method. The method should

receive two arguments: a `$notifiable` and a `$notification`.

Within the `send` method, you may call methods on the notification to retrieve a message object understood by your channel and then send the notification to the `$notifiable` instance however you wish:

```
<?php

namespace App\Notifications;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * Send the given notification.
     */
    public function send(object $notifiable, Notification $notification): void
    {
        $message = $notification->toVoice($notifiable);

        // Send notification to the $notifiable instance...
    }
}
```

Once your notification channel class has been defined, you may return the class name from the `via` method of any of your notifications. In this example, the `toVoice` method of your notification can return whatever object you choose to represent voice messages. For example, you might define your own `VoiceMessage` class to represent these messages:

```
<?php

namespace App\Notifications;

use App\Notifications\Messages\VoiceMessage;
use App\Notifications\VoiceChannel;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Get the notification channels.
     */
    public function via(object $notifiable): string
    {
        return VoiceChannel::class;
    }

    /**
     * Get the voice representation of the notification.
     */
    public function toVoice(object $notifiable): VoiceMessage
    {
        // ...
    }
}
```

Package Development

- [Introduction](#)
 - [A Note on Facades](#)
- [Package Discovery](#)
- [Service Providers](#)
- [Resources](#)
 - [Configuration](#)
 - [Migrations](#)
 - [Routes](#)
 - [Language Files](#)
 - [Views](#)
 - [View Components](#)
 - ["About" Artisan Command](#)
- [Commands](#)
- [Public Assets](#)
- [Publishing File Groups](#)

Introduction

Packages are the primary way of adding functionality to Laravel. Packages might be anything from a great way to work with dates like [Carbon](#) or a package that allows you to associate files with Eloquent models like Spatie's [Laravel Media Library](#).

There are different types of packages. Some packages are stand-alone, meaning they work with any PHP framework. Carbon and PHPUnit are examples of stand-alone packages. Any of these packages may be used with Laravel by requiring them in your `composer.json` file.

On the other hand, other packages are specifically intended for use with Laravel. These packages may have routes, controllers, views, and configuration specifically intended to enhance a Laravel application. This guide primarily covers the development of those packages that are Laravel specific.

A Note on Facades

When writing a Laravel application, it generally does not matter if you use contracts or facades since both provide essentially equal levels of testability. However, when writing packages, your package will not typically have access to all of Laravel's testing helpers. If you would like to be able to write your package tests as if the package were installed inside a typical Laravel application, you may use the [Orchestral Testbench](#) package.

Package Discovery

In a Laravel application's `config/app.php` configuration file, the `providers` option defines a list of service providers that should be loaded by Laravel. When someone installs your package, you will typically want your service provider to be included in this list. Instead of requiring users to manually add your service provider to the list, you may define the provider in the `extra` section of your package's `composer.json` file. In addition to service providers, you may also list any [facades](#) you would like to be registered:

```
"extra": {
    "laravel": {
        "providers": [
            "Barryvdh\\Debugbar\\ServiceProvider"
        ],
        "aliases": {
            "Debugbar": "Barryvdh\\Debugbar\\Facade"
        }
    }
},
```

Once your package has been configured for discovery, Laravel will automatically register its service providers and facades when it is installed, creating a convenient installation experience for your package's users.

Opting Out of Package Discovery

If you are the consumer of a package and would like to disable package discovery for a package, you may list the package name in the `extra` section of your application's `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

You may disable package discovery for all packages using the `*` character inside of your application's `dont-discover` directive:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
},
```

Service Providers

[Service providers](#) are the connection point between your package and Laravel. A service provider is responsible for binding things into Laravel's [service container](#) and informing Laravel where to load package resources such as views, configuration, and language files.

A service provider extends the `Illuminate\Support\ServiceProvider` class and contains two methods: `register` and `boot`. The base `ServiceProvider` class is located in the `illuminate/support` Composer package, which you should add to your own package's dependencies. To learn more about the structure and purpose of service providers, check out [their documentation](#).

Resources

Configuration

Typically, you will need to publish your package's configuration file to the application's `config` directory. This will allow users of your package to easily override your default configuration options. To allow your configuration files to be

published, call the `publishes` method from the `boot` method of your service provider:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../config/courier.php' => config_path('courier.php'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` command, your file will be copied to the specified publish location. Once your configuration has been published, its values may be accessed like any other configuration file:

```
$value = config('courier.option');
```



You should not define closures in your configuration files. They can not be serialized correctly when users execute the `config:cache` Artisan command.

Default Package Configuration

You may also merge your own package configuration file with the application's published copy. This will allow your users to define only the options they actually want to override in the published copy of the configuration file. To merge the configuration file values, use the `mergeConfigFrom` method within your service provider's `register` method.

The `mergeConfigFrom` method accepts the path to your package's configuration file as its first argument and the name of the application's copy of the configuration file as its second argument:

```
/**
 * Register any application services.
 */
public function register(): void
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```



This method only merges the first level of the configuration array. If your users partially define a multi-dimensional configuration array, the missing options will not be merged.

Routes

If your package contains routes, you may load them using the `loadRoutesFrom` method. This method will automatically determine if the application's routes are cached and will not load your routes file if the routes have already been cached:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadRoutesFrom(__DIR__.'/../routes/web.php');
}
```

Migrations

If your package contains [database migrations](#), you may use the `loadMigrationsFrom` method to inform Laravel how to load them. The `loadMigrationsFrom` method accepts the path to your package's migrations as its only argument:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
}
```

Once your package's migrations have been registered, they will automatically be run when the `php artisan migrate` command is executed. You do not need to export them to the application's `database/migrations` directory.

Language Files

If your package contains [language files](#), you may use the `loadTranslationsFrom` method to inform Laravel how to load them. For example, if your package is named `courier`, you should add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');
}
```

Package translation lines are referenced using the `package::file.line` syntax convention. So, you may load the `courier` package's `welcome` line from the `messages` file like so:

```
echo trans('courier::messages.welcome');
```

You can register JSON translation files for your package using the `loadJsonTranslationsFrom` method. This method accepts the path to the directory that contains your package's JSON translation files:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadJsonTranslationsFrom(__DIR__.'/../lang');
}
```

Publishing Language Files

If you would like to publish your package's language files to the application's `lang/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package paths and their desired publish locations. For example, to publish the language files for the `courier` package, you may do the following:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');

    $this->publishes([
        __DIR__.'/../lang' => $this->app->langPath('vendor/courier'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's language files will be published to the specified publish location.

Views

To register your package's `views` with Laravel, you need to tell Laravel where the views are located. You may do this using the service provider's `loadViewsFrom` method. The `loadViewsFrom` method accepts two arguments: the path to your view templates and your package's name. For example, if your package's name is `courier`, you would add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

Package views are referenced using the `package::view` syntax convention. So, once your view path is registered in a service provider, you may load the `dashboard` view from the `courier` package like so:

```
Route::get('/dashboard', function () {
    return view('courier::dashboard');
});
```

Overriding Package Views

When you use the `loadViewsFrom` method, Laravel actually registers two locations for your views: the application's `resources/views/vendor` directory and the directory you specify. So, using the `courier` package as an example, Laravel will first check if a custom version of the view has been placed in the `resources/views/vendor/courier` directory by the developer. Then, if the view has not been customized, Laravel will search the package view directory you specified in your call to `loadViewsFrom`. This makes it easy for package users to customize / override your package's views.

Publishing Views

If you would like to make your views available for publishing to the application's `resources/views/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package view paths

and their desired publish locations:

```
/**
 * Bootstrap the package services.
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

    $this->publishes([
        __DIR__.'/../resources/views' => resource_path('views/vendor/courier'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's views will be copied to the specified publish location.

View Components

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.
 */
public function boot(): void
{
    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 `Nightshade\Views\Components` namespace:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap your package's services.
 */
public function boot(): void
{
    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.

Anonymous Components

If your package contains anonymous components, they must be placed within a `components` directory of your package's "views" directory (as specified by the [loadViewsFrom](#) method). Then, you may render them by prefixing the component name with the package's view namespace:

```
<x-courier::alert />
```

"About" Artisan Command

Laravel's built-in `about` Artisan command provides a synopsis of the application's environment and configuration. Packages may push additional information to this command's output via the `AboutCommand` class. Typically, this information may be added from your package service provider's `boot` method:

```
use Illuminate\Foundation\Console\AboutCommand;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    AboutCommand::add('My Package', fn () => ['Version' => '1.0.0']);
}
```

Commands

To register your package's Artisan commands with Laravel, you may use the `commands` method. This method expects an array of command class names. Once the commands have been registered, you may execute them using the [Artisan CLI](#):

```
use Courier\Console\Commands\InstallCommand;
use Courier\Console\Commands\NetworkCommand;

/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            InstallCommand::class,
            NetworkCommand::class,
        ]);
    }
}
```

Public Assets

Your package may have assets such as JavaScript, CSS, and images. To publish these assets to the application's `public` directory, use the service provider's `publishes` method. In this example, we will also add a `public` asset group tag, which may be used to easily publish groups of related assets:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../public' => public_path('vendor/courier'),
    ], 'public');
}
```

Now, when your package's users execute the `vendor:publish` command, your assets will be copied to the specified publish location. Since users will typically need to overwrite the assets every time the package is updated, you may use the `--force` flag:

```
php artisan vendor:publish --tag=public --force
```

Publishing File Groups

You may want to publish groups of package assets and resources separately. For instance, you might want to allow your users to publish your package's configuration files without being forced to publish your package's assets. You may do this by "tagging" them when calling the `publishes` method from a package's service provider. For example, let's use tags to define two publish groups for the `courier` package (`courier-config` and `courier-migrations`) in the `boot` method of the package's service provider:

```
/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'courier-config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'courier-migrations');
}
```

Now your users may publish these groups separately by referencing their tag when executing the `vendor:publish` command:

```
php artisan vendor:publish --tag=courier-config
```

Processes

- [Introduction](#)
- [Invoking Processes](#)
 - [Process Options](#)
 - [Process Output](#)
 - [Pipelines](#)
- [Asynchronous Processes](#)
 - [Process IDs and Signals](#)
 - [Asynchronous Process Output](#)
- [Concurrent Processes](#)
 - [Naming Pool Processes](#)
 - [Pool Process IDs and Signals](#)
- [Testing](#)
 - [Faking Processes](#)
 - [Faking Specific Processes](#)
 - [Faking Process Sequences](#)
 - [Faking Asynchronous Process Lifecycles](#)
 - [Available Assertions](#)
 - [Preventing Stray Processes](#)

Introduction

Laravel provides an expressive, minimal API around the [Symfony Process component](#), allowing you to conveniently invoke external processes from your Laravel application. Laravel's process features are focused on the most common use cases and a wonderful developer experience.

Invoking Processes

To invoke a process, you may use the `run` and `start` methods offered by the `Process` facade. The `run` method will invoke a process and wait for the process to finish executing, while the `start` method is used for asynchronous process execution. We'll examine both approaches within this documentation. First, let's examine how to invoke a basic, synchronous process and inspect its result:

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

return $result->output();
```

Of course, the `Illuminate\Contracts\Process\ProcessResult` instance returned by the `run` method offers a variety of helpful methods that may be used to inspect the process result:

```
$result = Process::run('ls -la');

$result->successful();
$result->failed();
$result->exitCode();
$result->output();
$result->errorOutput();
```

Throwing Exceptions

If you have a process result and would like to throw an instance of `Illuminate\Process\Exceptions\ProcessFailedException` if the exit code is greater than zero (thus indicating failure), you may use the `throw` and `throwIf` methods. If the process did not fail, the process result instance will be returned:

```
$result = Process::run('ls -la')->throw();

$result = Process::run('ls -la')->throwIf($condition);
```

Process Options

Of course, you may need to customize the behavior of a process before invoking it. Thankfully, Laravel allows you to tweak a variety of process features, such as the working directory, timeout, and environment variables.

Working Directory Path

You may use the `path` method to specify the working directory of the process. If this method is not invoked, the process will inherit the working directory of the currently executing PHP script:

```
$result = Process::path(__DIR__)->run('ls -la');
```

Input

You may provide input via the "standard input" of the process using the `input` method:

```
$result = Process::input('Hello World')->run('cat');
```

Timeouts

By default, processes will throw an instance of `Illuminate\Process\Exceptions\ProcessTimedOutException` after executing for more than 60 seconds. However, you can customize this behavior via the `timeout` method:

```
$result = Process::timeout(120)->run('bash import.sh');
```

Or, if you would like to disable the process timeout entirely, you may invoke the `forever` method:

```
$result = Process::forever()->run('bash import.sh');
```

The `idleTimeout` method may be used to specify the maximum number of seconds the process may run without returning any output:

```
$result = Process::timeout(60)->idleTimeout(30)->run('bash import.sh');
```

Environment Variables

Environment variables may be provided to the process via the `env` method. The invoked process will also inherit all of the environment variables defined by your system:

```
$result = Process::forever()
    ->env(['IMPORT_PATH' => __DIR__])
    ->run('bash import.sh');
```

If you wish to remove an inherited environment variable from the invoked process, you may provide that environment variable with a value of `false`:

```
$result = Process::forever()
    ->env(['LOAD_PATH' => false])
    ->run('bash import.sh');
```

TTY Mode

The `tty` method may be used to enable TTY mode for your process. TTY mode connects the input and output of the process to the input and output of your program, allowing your process to open an editor like Vim or Nano as a process:

```
Process::forever()->tty()->run('vim');
```

Process Output

As previously discussed, process output may be accessed using the `output` (stdout) and `errorOutput` (stderr) methods on a process result:

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

echo $result->output();
echo $result->errorOutput();
```

However, output may also be gathered in real-time by passing a closure as the second argument to the `run` method. The closure will receive two arguments: the "type" of output (`stdout` or `stderr`) and the output string itself:

```
$result = Process::run('ls -la', function (string $type, string $output) {
    echo $output;
});
```

Laravel also offers the `seeInOutput` and `seeInErrorOutput` methods, which provide a convenient way to determine if a given string was contained in the process' output:

```
if (Process::run('ls -la')->seeInOutput('laravel')) {
    // ...
}
```

Disabling Process Output

If your process is writing a significant amount of output that you are not interested in, you can conserve memory by disabling output retrieval entirely. To accomplish this, invoke the `quietly` method while building the process:

```
use Illuminate\Support\Facades\Process;

$result = Process::quietly()->run('bash import.sh');
```

Pipelines

Sometimes you may want to make the output of one process the input of another process. This is often referred to as "piping" the output of a process into another. The `pipe` method provided by the `Process` facades makes this easy to

accomplish. The `pipe` method will execute the piped processes synchronously and return the process result for the last process in the pipeline:

```
use Illuminate\Process\Pipe;
use Illuminate\Support\Facades\Process;

$result = Process::pipe(function (Pipe $pipe) {
    $pipe->command('cat example.txt');
    $pipe->command('grep -i "laravel"');
});

if ($result->successful()) {
    // ...
}
```

If you do not need to customize the individual processes that make up the pipeline, you may simply pass an array of command strings to the `pipe` method:

```
$result = Process::pipe([
    'cat example.txt',
    'grep -i "laravel"',
]);
```

The process output may be gathered in real-time by passing a closure as the second argument to the `pipe` method. The closure will receive two arguments: the "type" of output (`stdout` or `stderr`) and the output string itself:

```
$result = Process::pipe(function (Pipe $pipe) {
    $pipe->command('cat example.txt');
    $pipe->command('grep -i "laravel"');
}, function (string $type, string $output) {
    echo $output;
});
```

Laravel also allows you to assign string keys to each process within a pipeline via the `as` method. This key will also be passed to the output closure provided to the `pipe` method, allowing you to determine which process the output belongs to:

```
$result = Process::pipe(function (Pipe $pipe) {
    $pipe->as('first')->command('cat example.txt');
    $pipe->as('second')->command('grep -i "laravel"');
})->start(function (string $type, string $output, string $key) {
    // ...
});
```

Asynchronous Processes

While the `run` method invokes processes synchronously, the `start` method may be used to invoke a process asynchronously. This allows your application to continue performing other tasks while the process runs in the background. Once the process has been invoked, you may utilize the `running` method to determine if the process is still running:

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    // ...
}

$result = $process->wait();
```

As you may have noticed, you may invoke the `wait` method to wait until the process is finished executing and retrieve the process result instance:

```
$process = Process::timeout(120)->start('bash import.sh');

// ...

$result = $process->wait();
```

Process IDs and Signals

The `id` method may be used to retrieve the operating system assigned process ID of the running process:

```
$process = Process::start('bash import.sh');

return $process->id();
```

You may use the `signal` method to send a "signal" to the running process. A list of predefined signal constants can be found within the [PHP documentation](#):

```
$process->signal(SIGUSR2);
```

Asynchronous Process Output

While an asynchronous process is running, you may access its entire current output using the `output` and `errorOutput` methods; however, you may utilize the `latestOutput` and `latestErrorOutput` to access the output from the process that has occurred since the output was last retrieved:

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    echo $process->latestOutput();
    echo $process->latestErrorOutput();

    sleep(1);
}
```

Like the `run` method, output may also be gathered in real-time from asynchronous processes by passing a closure as the second argument to the `start` method. The closure will receive two arguments: the "type" of output (`stdout` or `stderr`) and the output string itself:

```
$process = Process::start('bash import.sh', function (string $type, string $output) {
    echo $output;
});

$result = $process->wait();
```

Concurrent Processes

Laravel also makes it a breeze to manage a pool of concurrent, asynchronous processes, allowing you to easily execute many tasks simultaneously. To get started, invoke the `pool` method, which accepts a closure that receives an instance of `Illuminate\Process\Pool`.

Within this closure, you may define the processes that belong to the pool. Once a process pool is started via the `start` method, you may access the `collection` of running processes via the `running` method:

```
use Illuminate\Process\Pool;
use Illuminate\Support\Facades\Process;

$pool = Process::pool(function (Pool $pool) {
    $pool->path(__DIR__)->command('bash import-1.sh');
    $pool->path(__DIR__)->command('bash import-2.sh');
    $pool->path(__DIR__)->command('bash import-3.sh');
})->start(function (string $type, string $output, int $key) {
    // ...
});

while ($pool->running()->isNotEmpty()) {
    // ...
}

$results = $pool->wait();
```

As you can see, you may wait for all of the pool processes to finish executing and resolve their results via the `wait` method. The `wait` method returns an array accessible object that allows you to access the process result instance of each process in the pool by its key:

```
$results = $pool->wait();

echo $results[0]->output();
```

Or, for convenience, the `concurrently` method may be used to start an asynchronous process pool and immediately wait on its results. This can provide particularly expressive syntax when combined with PHP's array destructuring capabilities:

```
[$first, $second, $third] = Process::concurrently(function (Pool $pool) {
    $pool->path(__DIR__)->command('ls -la');
    $pool->path(app_path())->command('ls -la');
    $pool->path(storage_path())->command('ls -la');
});

echo $first->output();
```

Naming Pool Processes

Accessing process pool results via a numeric key is not very expressive; therefore, Laravel allows you to assign string keys to each process within a pool via the `as` method. This key will also be passed to the closure provided to the `start` method, allowing you to determine which process the output belongs to:

```
$pool = Process::pool(function (Pool $pool) {
    $pool->as('first')->command('bash import-1.sh');
    $pool->as('second')->command('bash import-2.sh');
    $pool->as('third')->command('bash import-3.sh');
})->start(function (string $type, string $output, string $key) {
    // ...
});

$results = $pool->wait();

return $results['first']->output();
```

Pool Process IDs and Signals

Since the process pool's `running` method provides a collection of all invoked processes within the pool, you may easily access the underlying pool process IDs:

```
$processIds = $pool->running()->each->id();
```

And, for convenience, you may invoke the `signal` method on a process pool to send a signal to every process within the pool:

```
$pool->signal(SIGUSR2);
```

Testing

Many Laravel services provide functionality to help you easily and expressively write tests, and Laravel's process service is no exception. The `Process` facade's `fake` method allows you to instruct Laravel to return stubbed / dummy results when processes are invoked.

Faking Processes

To explore Laravel's ability to fake processes, let's imagine a route that invokes a process:

```
use Illuminate\Support\Facades\Process;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
    Process::run('bash import.sh');

    return 'Import complete!';
});
```

When testing this route, we can instruct Laravel to return a fake, successful process result for every invoked process by calling the `fake` method on the `Process` facade with no arguments. In addition, we can even `assert` that a given process was "run":

```
<?php

namespace Tests\Feature;

use Illuminate\Process\PendingProcess;
use Illuminate\Contracts\Process\ProcessResult;
use Illuminate\Support\Facades\Process;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_process_is_invoked(): void
    {
        Process::fake();

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

        // Simple process assertion...
        Process::assertRan('bash import.sh');

        // Or, inspecting the process configuration...
        Process::assertRan(function (PendingProcess $process, ProcessResult $result) {
            return $process->command === 'bash import.sh' &&
                $process->timeout === 60;
        });
    }
}
```

As discussed, invoking the `fake` method on the `Process` facade will instruct Laravel to always return a successful process result with no output. However, you may easily specify the output and exit code for faked processes using the `Process` facade's `result` method:

```
Process::fake([
    '*' => Process::result(
        output: 'Test output',
        errorOutput: 'Test error output',
        exitCode: 1,
    ),
]);
```

Faking Specific Processes

As you may have noticed in a previous example, the `Process` facade allows you to specify different fake results per process by passing an array to the `fake` method.

The array's keys should represent command patterns that you wish to fake and their associated results. The `*` character may be used as a wildcard character. Any process commands that have not been faked will actually be invoked. You may use the `Process` facade's `result` method to construct stub / fake results for these commands:

```
Process::fake([
    'cat *' => Process::result(
        output: 'Test "cat" output',
    ),
    'ls *' => Process::result(
        output: 'Test "ls" output',
    ),
]);

```

If you do not need to customize the exit code or error output of a faked process, you may find it more convenient to specify the fake process results as simple strings:

```
Process::fake([
    'cat *' => 'Test "cat" output',
    'ls *' => 'Test "ls" output',
]);

```

Faking Process Sequences

If the code you are testing invokes multiple processes with the same command, you may wish to assign a different fake process result to each process invocation. You may accomplish this via the `Process` facade's `sequence` method:

```
Process::fake([
    'ls *' => Process::sequence()
        ->push(Process::result('First invocation'))
        ->push(Process::result('Second invocation')),
]);

```

Faking Asynchronous Process Lifecycles

Thus far, we have primarily discussed faking processes which are invoked synchronously using the `run` method. However, if you are attempting to test code that interacts with asynchronous processes invoked via `start`, you may need a more sophisticated approach to describing your fake processes.

For example, let's imagine the following route which interacts with an asynchronous process:

```
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
    $process = Process::start('bash import.sh');

    while ($process->running()) {
        Log::info($process->latestOutput());
        Log::info($process->latestErrorOutput());
    }

    return 'Done';
});

```

To properly fake this process, we need to be able to describe how many times the `running` method should return `true`. In addition, we may want to specify multiple lines of output that should be returned in sequence. To accomplish this, we can use the `Process` facade's `describe` method:

```
Process::fake([
    'bash import.sh' => Process::describe()
        ->output('First line of standard output')
        ->errorOutput('First line of error output')
        ->output('Second line of standard output')
        ->exitCode(0)
        ->iterations(3),
]);

```

Let's dig into the example above. Using the `output` and `errorOutput` methods, we may specify multiple lines of output that will be returned in sequence. The `exitCode` method may be used to specify the final exit code of the fake process. Finally, the `iterations` method may be used to specify how many times the `running` method should return `true`.

Available Assertions

As [previously discussed](#), Laravel provides several process assertions for your feature tests. We'll discuss each of these assertions below.

`assertRan`

Assert that a given process was invoked:

```
use Illuminate\Support\Facades\Process;

Process::assertRan('ls -la');
```

The `assertRan` method also accepts a closure, which will receive an instance of a process and a process result, allowing you to inspect the process' configured options. If this closure returns `true`, the assertion will "pass":

```
Process::assertRan(fn ($process, $result) =>
    $process->command === 'ls -la' &&
    $process->path === __DIR__ &&
    $process->timeout === 60
);
```

The `$process` passed to the `assertRan` closure is an instance of `Illuminate\Process\PendingProcess`, while the `$result` is an instance of `Illuminate\Contracts\Process\ProcessResult`.

`assertDidntRun`

Assert that a given process was not invoked:

```
use Illuminate\Support\Facades\Process;

Process::assertDidntRun('ls -la');
```

Like the `assertRan` method, the `assertDidntRun` method also accepts a closure, which will receive an instance of a process and a process result, allowing you to inspect the process' configured options. If this closure returns `true`, the assertion will "fail":

```
Process::assertDidntRun(fn (PendingProcess $process, ProcessResult $result) =>
    $process->command === 'ls -la'
);
```

assertRanTimes

Assert that a given process was invoked a given number of times:

```
use Illuminate\Support\Facades\Process;

Process::assertRanTimes('ls -la', times: 3);
```

The `assertRanTimes` method also accepts a closure, which will receive an instance of a process and a process result, allowing you to inspect the process' configured options. If this closure returns `true` and the process was invoked the specified number of times, the assertion will "pass":

```
Process::assertRanTimes(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'ls -la';
}, times: 3);
```

Preventing Stray Processes

If you would like to ensure that all invoked processes have been faked throughout your individual test or complete test suite, you can call the `preventStrayProcesses` method. After calling this method, any processes that do not have a corresponding fake result will throw an exception rather than starting an actual process:

```
use Illuminate\Support\Facades\Process;

Process::preventStrayProcesses();

Process::fake([
    'ls *' => 'Test output...',
]);

// Fake response is returned...
Process::run('ls -la');

// An exception is thrown...
Process::run('bash import.sh');
```

Queues

- [Introduction](#)
 - [Connections vs. Queues](#)
 - [Driver Notes and Prerequisites](#)
- [Creating Jobs](#)
 - [Generating Job Classes](#)
 - [Class Structure](#)
 - [Unique Jobs](#)
 - [Encrypted Jobs](#)
- [Job Middleware](#)
 - [Rate Limiting](#)
 - [Preventing Job Overlaps](#)
 - [Throttling Exceptions](#)
- [Dispatching Jobs](#)
 - [Delayed Dispatching](#)
 - [Synchronous Dispatching](#)
 - [Jobs & Database Transactions](#)
 - [Job Chaining](#)
 - [Customizing The Queue and Connection](#)
 - [Specifying Max Job Attempts / Timeout Values](#)
 - [Error Handling](#)
- [Job Batching](#)
 - [Defining Batchable Jobs](#)
 - [Dispatching Batches](#)
 - [Chains and Batches](#)
 - [Adding Jobs to Batches](#)
 - [Inspecting Batches](#)
 - [Cancelling Batches](#)
 - [Batch Failures](#)
 - [Pruning Batches](#)
 - [Storing Batches in DynamoDB](#)
- [Queueing Closures](#)
- [Running the Queue Worker](#)
 - [The `queue:work` Command](#)
 - [Queue Priorities](#)
 - [Queue Workers and Deployment](#)
 - [Job Expirations and Timeouts](#)
- [Supervisor Configuration](#)
- [Dealing With Failed Jobs](#)
 - [Cleaning Up After Failed Jobs](#)
 - [Retrying Failed Jobs](#)
 - [Ignoring Missing Models](#)
 - [Pruning Failed Jobs](#)
 - [Storing Failed Jobs in DynamoDB](#)
 - [Disabling Failed Job Storage](#)
 - [Failed Job Events](#)
- [Clearing Jobs From Queues](#)
- [Monitoring Your Queues](#)
- [Testing](#)
 - [Faking a Subset of Jobs](#)

- [Testing Job Chains](#)
- [Testing Job Batches](#)
- [Job Events](#)

Introduction

While building your web application, you may have some tasks, such as parsing and storing an uploaded CSV file, that take too long to perform during a typical web request. Thankfully, Laravel allows you to easily create queued jobs that may be processed in the background. By moving time intensive tasks to a queue, your application can respond to web requests with blazing speed and provide a better user experience to your customers.

Laravel queues provide a unified queueing API across a variety of different queue backends, such as [Amazon SQS](#), [Redis](#), or even a relational database.

Laravel's queue configuration options are stored in your application's `config/queue.php` configuration file. In this file, you will find connection configurations for each of the queue drivers that are included with the framework, including the database, [Amazon SQS](#), [Redis](#), and [Beanstalkd](#) drivers, as well as a synchronous driver that will execute jobs immediately (for use during local development). A `null` queue driver is also included which discards queued jobs.



Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full [Horizon documentation](#) for more information.

Connections vs. Queues

Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". In your `config/queue.php` configuration file, there is a `connections` configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.

Note that each connection configuration example in the `queue` configuration file contains a `queue` attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the `queue` attribute of the connection configuration:

```
use App\Jobs\ProcessPodcast;

// This job is sent to the default connection's default queue...
ProcessPodcast::dispatch();

// This job is sent to the default connection's "emails" queue...
ProcessPodcast::dispatch()->onQueue('emails');
```

Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a `high` queue, you may run a worker that gives them higher processing priority:

```
php artisan queue:work --queue=high,default
```

Driver Notes and Prerequisites

Database

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

```
php artisan queue:table
php artisan migrate
```

Finally, don't forget to instruct your application to use the `database` driver by updating the `QUEUE_CONNECTION` variable in your application's `.env` file:

```
QUEUE_CONNECTION=database
```

Redis

In order to use the `redis` queue driver, you should configure a Redis database connection in your `config/database.php` configuration file.



The `serializer` and `compression` Redis options are not supported by the `redis` queue driver.

Redis Cluster

If your Redis queue connection uses a Redis Cluster, your queue names must contain a `key_hash` tag. This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

Blocking

When using the Redis queue, you may use the `block_for` configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.

Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to `5` to indicate that the driver should block for five seconds while waiting for a job to become available:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```



Setting `block_for` to `0` will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as `SIGTERM` from being handled until the next job has been processed.

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` or `phpredis` PHP extension

Creating Jobs

Generating Job Classes

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. If the `app/Jobs` directory doesn't exist, it will be created when you run the `make:job` Artisan command:

```
php artisan make:job ProcessPodcast
```

The generated class will implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating to Laravel that the job should be pushed onto the queue to run asynchronously.



Job stubs may be customized using [stub publishing](#).

Class Structure

Job classes are very simple, normally containing only a `handle` method that is invoked when the job is processed by the queue. To get started, let's take a look at an example job class. In this example, we'll pretend we manage a podcast publishing service and need to process the uploaded podcast files before they are published:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * Execute the job.
     */
    public function handle(AudioProcessor $processor): void
    {
        // Process uploaded podcast...
    }
}
```

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

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

handle Method Dependency Injection

The `handle` method is invoked when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Laravel [service container](#) automatically injects these dependencies.

If you would like to take total control over how the container injects dependencies into the `handle` method, you may use the container's `bindMethod` method. The `bindMethod` method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the `handle` method however you wish. Typically, you should call this method from the `boot` method of your [App\Providers\AppServiceProvider](#) [service provider](#):

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;
use Illuminate\Contracts\Foundation\Application;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function (ProcessPodcast $job, Application $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```



Binary data, such as raw image contents, should be passed through the `base64_encode` function before being passed to a queued job. Otherwise, the job may not properly serialize to JSON when being placed on the queue.

Queued Relationships

Because all loaded Eloquent model relationships also get serialized when a job is queued, the serialized job string can sometimes become quite large. Furthermore, when a job is deserialized and model relationships are re-retrieved from the database, they will be retrieved in their entirety. Any previous relationship constraints that were applied before the model was serialized during the job queueing process will not be applied when the job is deserialized. Therefore, if you wish to work with a subset of a given relationship, you should re-constrain that relationship within your queued job.

Or, to prevent relations from being serialized, you can call the `withoutRelations` method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

```
/**
 * Create a new job instance.
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

If you are using PHP constructor property promotion and would like to indicate that an Eloquent model should not have its relations serialized, you may use the `WithoutRelations` attribute:

```
use Illuminate\Queue\Attributes\WithoutRelations;

/**
 * Create a new job instance.
 */
public function __construct(
    #[WithoutRelations]
    public Podcast $podcast
) {
```

If a job receives a collection or array of Eloquent models instead of a single model, the models within that collection will not have their relationships restored when the job is deserialized and executed. This is to prevent excessive resource usage on jobs that deal with large numbers of models.

Unique Jobs



Unique jobs require a cache driver that supports [locks](#). Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks. In addition, unique job constraints do not apply to jobs within batches.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the `ShouldBeUnique` interface on your job class. This interface does not require you to define any additional methods on your class:

```
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

In the example above, the `UpdateSearchIndex` job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing.

In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

```
<?php

use App\Models\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    /**
     * The product instance.
     *
     * @var \App\Product
     */
    public $product;

    /**
     * The number of seconds after which the job's unique lock will be released.
     *
     * @var int
     */
    public $uniqueFor = 3600;

    /**
     * Get the unique ID for the job.
     */
    public function uniqueId(): string
    {
        return $this->product->id;
    }
}
```

In the example above, the `UpdateSearchIndex` job is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.



If your application dispatches jobs from multiple web servers or containers, you should ensure that all of your servers are communicating with the same central cache server so that Laravel can accurately determine if a job is unique.

Keeping Jobs Unique Until Processing Begins

By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

```
<?php

use App\Models\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    // ...
}
```

Unique Job Locks

Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a `lock` with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method that returns the cache driver that should be used:

```
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Support\Facades\Cache;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...

    /**
     * Get the cache driver for the unique job lock.
     */
    public function uniqueVia(): Repository
    {
        return Cache::driver('redis');
    }
}
```



If you only need to limit the concurrent processing of a job, use the `WithoutOverlapping` job middleware instead.

Encrypted Jobs

Laravel allows you to ensure the privacy and integrity of a job's data via `encryption`. To get started, simply add the `ShouldBeEncrypted` interface to the job class. Once this interface has been added to the class, Laravel will automatically encrypt your job before pushing it onto a queue:

```
<?php

use Illuminate\Contracts\Queue\ShouldBeEncrypted;
use Illuminate\Contracts\Queue\ShouldQueue;

class UpdateSearchIndex implements ShouldQueue, ShouldBeEncrypted
{
    // ...
}
```

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following `handle` method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

```
use Illuminate\Support\Facades\Redis;

/**
 * Execute the job.
 */
public function handle(): void
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
        info('Lock obtained...');

        // Handle job...
    }, function () {
        // Could not obtain lock...

        return $this->release(5);
    });
}
```

While this code is valid, the implementation of the `handle` method becomes noisy since it is cluttered with Redis rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.

Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in an `app/Jobs/Middleware` directory:

```
<?php

namespace App\Jobs\Middleware;

use Closure;
use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * Process the queued job.
     *
     * @param \Closure(object): void $next
     */
    public function handle(object $job, Closure $next): void
    {
        Redis::throttle('key')
            ->block(0)->allow(1)->every(5)
            ->then(function () use ($job, $next) {
                // Lock obtained...

                $next($job);
            }, function () use ($job) {
                // Could not obtain lock...

                $job->release(5);
            });
    }
}
```

As you can see, like [route middleware](#), job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

After creating job middleware, they may be attached to a job by returning them from the job's `middleware` method. This method does not exist on jobs scaffolded by the `make:job` Artisan command, so you will need to manually add it to your job class:

```
use App\Jobs\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited];
}
```



Job middleware can also be assigned to queueable event listeners, mailables, and notifications.

Rate Limiting

Although we just demonstrated how to write your own rate limiting job middleware, Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like [route rate limiters](#), job rate limiters are defined using the `RateLimiter` facade's `for` method.

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a `RateLimiter` in the `boot` method of your `AppServiceProvider`:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    RateLimiter::for('backups', function (object $job) {
        return $job->user->vipCustomer()
            ? Limit::none()
            : Limit::perHour(1)->by($job->user->id);
    });
}
```

In the example above, we defined an hourly rate limit; however, you may easily define a rate limit based on minutes using the `perMinute` method. In addition, you may pass any value you wish to the `by` method of the rate limit; however, this value is most often used to segment rate limits by customer:

```
return Limit::perMinute(50)->by($job->user->id);
```

Once you have defined your rate limit, you may attach the rate limiter to your job using the `Illuminate\Queue\Middleware\RateLimited` middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited('backups')];
}
```

Releasing a rate limited job back onto the queue will still increment the job's total number of `attempts`. You may wish to tune your `tries` and `maxExceptions` properties on your job class accordingly. Or, you may wish to use the [retryUntil](#) method to define the amount of time until the job should no longer be attempted.

If you do not want a job to be retried when it is rate limited, you may use the `dontRelease` method:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new RateLimited('backups'))->dontRelease()];
}
```



If you are using Redis, you may use the `Illuminate\Queue\Middleware\RateLimitedWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

Preventing Job Overlaps

Laravel includes an `Illuminate\Queue\Middleware\WithoutOverlapping` middleware that allows you to prevent job overlaps based on an arbitrary key. This can be helpful when a queued job is modifying a resource that should only be modified by one job at a time.

For example, let's imagine you have a queued job that updates a user's credit score and you want to prevent credit score update job overlaps for the same user ID. To accomplish this, you can return the `WithoutOverlapping` middleware from your job's `middleware` method:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new WithoutOverlapping($this->user->id)];
}
```

Any overlapping jobs of the same type will be released back to the queue. You may also specify the number of seconds that must elapse before the released job will be attempted again:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the `dontRelease` method:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(\new WithoutOverlapping($this->order->id))->dontRelease()];
}
```

The `WithoutOverlapping` middleware is powered by Laravel's atomic lock feature. Sometimes, your job may unexpectedly fail or timeout in such a way that the lock is not released. Therefore, you may explicitly define a lock expiration time using the `expireAfter` method. For example, the example below will instruct Laravel to release the `WithoutOverlapping` lock three minutes after the job has started processing:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(\new WithoutOverlapping($this->order->id))->expireAfter(180)];
}
```



The `WithoutOverlapping` middleware requires a cache driver that supports `locks`. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks.

Sharing Lock Keys Across Job Classes

By default, the `WithoutOverlapping` middleware will only prevent overlapping jobs of the same class. So, although two different job classes may use the same lock key, they will not be prevented from overlapping. However, you can instruct Laravel to apply the key across job classes using the `shared` method:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

class ProviderIsDown
{
    // ...

    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}

class ProviderIsUp
{
    // ...

    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}
```

Throttling Exceptions

Laravel includes a `Illuminate\Queue\Middleware\ThrottlesExceptions` middleware that allows you to throttle exceptions. Once the job throws a given number of exceptions, all further attempts to execute the job are delayed until a specified time interval lapses. This middleware is particularly useful for jobs that interact with third-party services that are unstable.

For example, let's imagine a queued job that interacts with a third-party API that begins throwing exceptions. To throttle exceptions, you can return the `ThrottlesExceptions` middleware from your job's `middleware` method. Typically, this middleware should be paired with a job that implements [time based attempts](#):

```

use DateTime;
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new ThrottlesExceptions(10, 5)];
}

/**
 * Determine the time at which the job should timeout.
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(5);
}

```

The first constructor argument accepted by the middleware is the number of exceptions the job can throw before being throttled, while the second constructor argument is the number of minutes that should elapse before the job is attempted again once it has been throttled. In the code example above, if the job throws 10 exceptions within 5 minutes, we will wait 5 minutes before attempting the job again.

When a job throws an exception but the exception threshold has not yet been reached, the job will typically be retried immediately. However, you may specify the number of minutes such a job should be delayed by calling the `backoff` method when attaching the middleware to the job:

```

use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 5))>backoff(5)];
}

```

Internally, this middleware uses Laravel's cache system to implement rate limiting, and the job's class name is utilized as the cache "key". You may override this key by calling the `by` method when attaching the middleware to your job. This may be useful if you have multiple jobs interacting with the same third-party service and you would like them to share a common throttling "bucket":

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 10))>by('key')];
}
```



If you are using Redis, you may use the `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic exception throttling middleware.

Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` method on the job itself. The arguments passed to the `dispatch` method will be given to the job's constructor:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast);

        return redirect('/podcasts');
    }
}
```

If you would like to conditionally dispatch a job, you may use the `dispatchIf` and `dispatchUnless` methods:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);

ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

In new Laravel applications, the `sync` driver is the default queue driver. This driver executes jobs synchronously in the foreground of the current request, which is often convenient during local development. If you would like to actually begin queueing jobs for background processing, you may specify a different queue driver within your application's `config/queue.php` configuration file.

Delayed Dispatching

If you would like to specify that a job should not be immediately available for processing by a queue worker, you may use the `delay` method when dispatching the job. For example, let's specify that a job should not be available for processing until 10 minutes after it has been dispatched:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast)
            ->delay(now() -> addMinutes(10));

        return redirect('/podcasts');
    }
}
```



The Amazon SQS queue service has a maximum delay time of 15 minutes.

Dispatching After the Response is Sent to the Browser

Alternatively, the `dispatchAfterResponse` method delays dispatching a job until after the HTTP response is sent to the user's browser if your web server is using FastCGI. This will still allow the user to begin using the application even though a queued job is still executing. This should typically only be used for jobs that take about a second, such as sending an email. Since they are processed within the current HTTP request, jobs dispatched in this fashion do not require a queue worker to be running in order for them to be processed:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

You may also `dispatch` a closure and chain the `afterResponse` method onto the `dispatch` helper to execute a closure after the HTTP response has been sent to the browser:

```
use App\Mail>WelcomeMessage;
use Illuminate\Support\Facades\Mail;

dispatch(function () {
    Mail::to('taylor@example.com')->send(new WelcomeMessage());
})->afterResponse();
```

Synchronous Dispatching

If you would like to dispatch a job immediately (synchronously), you may use the `dispatchSync` method. When using this method, the job will not be queued and will be executed immediately within the current process:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Create podcast...

        ProcessPodcast::dispatchSync($podcast);

        return redirect('/podcasts');
    }
}
```

Jobs & Database Transactions

While it is perfectly fine to dispatch jobs within database transactions, you should take special care to ensure that your job will actually be able to execute successfully. When dispatching a job within a transaction, it is possible that the job will be processed by a worker before the parent transaction has committed. When this happens, any updates you have made to models or database records during the database transaction(s) may not yet be reflected in the database. In addition, any models or database records created within the transaction(s) may not exist in the database.

Thankfully, Laravel provides several methods of working around this problem. First, you may set the `after_commit` connection option in your queue connection's configuration array:

```
'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],
```

When the `after_commit` option is `true`, you may dispatch jobs within database transactions; however, Laravel will wait until the open parent database transactions have been committed before actually dispatching the job. Of course, if no database transactions are currently open, the job will be dispatched immediately.

If a transaction is rolled back due to an exception that occurs during the transaction, the jobs that were dispatched during that transaction will be discarded.



Setting the `after_commit` configuration option to `true` will also cause any queued event listeners, mailables, notifications, and broadcast events to be dispatched after all open database transactions have been committed.

Specifying Commit Dispatch Behavior Inline

If you do not set the `after_commit` queue connection configuration option to `true`, you may still indicate that a specific job should be dispatched after all open database transactions have been committed. To accomplish this, you may chain the `afterCommit` method onto your dispatch operation:

```
use App\Jobs\ProcessPodcast;

ProcessPodcast::dispatch($podcast)->afterCommit();
```

Likewise, if the `after_commit` configuration option is set to `true`, you may indicate that a specific job should be dispatched immediately without waiting for any open database transactions to commit:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

Job Chaining

Job chaining allows you to specify a list of queued jobs that should be run in sequence after the primary job has executed successfully. If one job in the sequence fails, the rest of the jobs will not be run. To execute a queued job chain, you may use the `chain` method provided by the `Bus` facade. Laravel's command bus is a lower level component that queued job dispatching is built on top of:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->dispatch();
```

In addition to chaining job class instances, you may also chain closures:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    function () {
        Podcast::update(/* ... */);
    },
])->dispatch();
```



Deleting jobs using the `$this->delete()` method within the job will not prevent chained jobs from being processed. The chain will only stop executing if a job in the chain fails.

Chain Connection and Queue

If you would like to specify the connection and queue that should be used for the chained jobs, you may use the `onConnection` and `onQueue` methods. These methods specify the queue connection and queue name that should be used unless the queued job is explicitly assigned a different connection / queue:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

Chain Failures

When chaining jobs, you may use the `catch` method to specify a closure that should be invoked if a job within the chain fails. The given callback will receive the `Throwable` instance that caused the job failure:

```
use Illuminate\Support\Facades\Bus;
use Throwable;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->catch(function (Throwable $e) {
    // A job within the chain has failed...
})->dispatch();
```



Since chain callbacks are serialized and executed at a later time by the Laravel queue, you should not use the `$this` variable within chain callbacks.

Customizing The Queue a Connection

Dispatching to a Particular Queue

By pushing jobs to different queues, you may "categorize" your queued jobs and even prioritize how many workers you assign to various queues. Keep in mind, this does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method when dispatching the job:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onQueue('processing');

        return redirect('/podcasts');
    }
}
```

Alternatively, you may specify the job's queue by calling the `onQueue` method within the job's constructor:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct()
    {
        $this->onQueue('processing');
    }
}
```

Dispatching to a Particular Connection

If your application interacts with multiple queue connections, you may specify which connection to push a job to using the `onConnection` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onConnection('sqS');

        return redirect('/podcasts');
    }
}
```

You may chain the `onConnection` and `onQueue` methods together to specify the connection and the queue for a job:

```
ProcessPodcast::dispatch($podcast)
    ->onConnection('sq')
    ->onQueue('processing');
```

Alternatively, you may specify the job's connection by calling the `onConnection` method within the job's constructor:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct()
    {
        $this->onConnection('sq');
    }
}
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

If one of your queued jobs is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a job may be attempted.

One approach to specifying the maximum number of times a job may be attempted is via the `--tries` switch on the Artisan command line. This will apply to all jobs processed by the worker unless the job being processed specifies the number of times it may be attempted:

```
php artisan queue:work --tries=3
```

If a job exceeds its maximum number of attempts, it will be considered a "failed" job. For more information on handling failed jobs, consult the [failed job documentation](#). If `--tries=0` is provided to the `queue:work` command, the job will be retried indefinitely.

You may take a more granular approach by defining the maximum number of times a job may be attempted on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the `--tries` value provided on the command line:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

If you need dynamic control over a particular job's maximum attempts, you may define a `tries` method on the job:

```
/*
 * Determine number of times the job may be attempted.
 */
public function tries(): int
{
    return 5;
}
```

Time Based Attempts

As an alternative to defining how many times a job may be attempted before it fails, you may define a time at which the job should no longer be attempted. This allows a job to be attempted any number of times within a given time frame. To define the time at which a job should no longer be attempted, add a `retryUntil` method to your job class. This method should return a `DateTime` instance:

```
use DateTime;

/**
 * Determine the time at which the job should timeout.
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(10);
}
```



You may also define a `tries` property or `retryUntil` method on your [queued event listeners](#).

Max Exceptions

Sometimes you may wish to specify that a job may be attempted many times, but should fail if the retries are triggered by a given number of unhandled exceptions (as opposed to being released by the `release` method directly). To accomplish this, you may define a `maxExceptions` property on your job class:

```
<?php

namespace App\Jobs;

use Illuminate\Support\Facades\Redis;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 25;

    /**
     * The maximum number of unhandled exceptions to allow before failing.
     *
     * @var int
     */
    public $maxExceptions = 3;

    /**
     * Execute the job.
     */
    public function handle(): void
    {
        Redis::throttle('key')->allow(10)->every(60)->then(function () {
            // Lock obtained, process the podcast...
        }, function () {
            // Unable to obtain lock...
            return $this->release(10);
        });
    }
}
```

In this example, the job is released for ten seconds if the application is unable to obtain a Redis lock and will continue to be retried up to 25 times. However, the job will fail if three unhandled exceptions are thrown by the job.

Timeout

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. By default, the timeout value is 60 seconds. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a [process manager configured on your server](#).

The maximum number of seconds that jobs can run may be specified using the `--timeout` switch on the Artisan command line:

```
php artisan queue:work --timeout=30
```

If the job exceeds its maximum attempts by continually timing out, it will be marked as failed.

You may also define the maximum number of seconds a job should be allowed to run on the job class itself. If the timeout is specified on the job, it will take precedence over any timeout specified on the command line:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

Sometimes, IO blocking processes such as sockets or outgoing HTTP connections may not respect your specified timeout. Therefore, when using these features, you should always attempt to specify a timeout using their APIs as well. For example, when using Guzzle, you should always specify a connection and request timeout value.



The `pcntl` PHP extension must be installed in order to specify job timeouts. In addition, a job's "timeout" value should always be less than its `"retry after"` value. Otherwise, the job may be re-attempted before it has actually finished executing or timed out.

Failing on Timeout

If you would like to indicate that a job should be marked as [failed](#) on timeout, you may define the `$failOnTimeout` property on the job class:

```
/**
 * Indicate if the job should be marked as failed on timeout.
 *
 * @var bool
 */
public $failOnTimeout = true;
```

Error Handling

If an exception is thrown while the job is being processed, the job will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The maximum number of attempts is defined by the `--tries` switch used on the `queue:work` Artisan command. Alternatively, the maximum number of attempts may be defined on the job class itself. More information on running the queue worker [can be found below](#).

Manually Releasing a Job

Sometimes you may wish to manually release a job back onto the queue so that it can be attempted again at a later time. You may accomplish this by calling the `release` method:

```
/**
 * Execute the job.
 */
public function handle(): void
{
    // ...

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

By default, the `release` method will release the job back onto the queue for immediate processing. However, you may instruct the queue to not make the job available for processing until a given number of seconds has elapsed by passing an integer or date instance to the `release` method:

```
$this->release(10);

$this->release(now()->addSeconds(10));
```

Manually Failing a Job

Occasionally you may need to manually mark a job as "failed". To do so, you may call the `fail` method:

```
/**
 * Execute the job.
 */
public function handle(): void
{
    // ...

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

If you would like to mark your job as failed because of an exception that you have caught, you may pass the exception to the `fail` method. Or, for convenience, you may pass a string error message which will be converted to an exception for you:

```
$this->fail($exception);

$this->fail('Something went wrong.');
```



For more information on failed jobs, check out the [documentation on dealing with job failures](#).

Job Batching

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing. Before getting started, you should create a database migration to build a table which will contain meta information about your job batches, such as their completion percentage. This migration may be generated using the `queue:batches-table` Artisan command:

```
php artisan queue:batches-table
```

```
php artisan migrate
```

Defining Batchable Jobs

To define a batchable job, you should [create a queueable job](#) as normal; however, you should add the `Illuminate\Bus\Batchable` trait to the job class. This trait provides access to a `batch` method which may be used to retrieve the current batch that the job is executing within:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
    use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Execute the job.
     */
    public function handle(): void
    {
        if ($this->batch()->cancelled()) {
            // Determine if the batch has been cancelled...

            return;
        }

        // Import a portion of the CSV file...
    }
}
```

Dispatching Batches

To dispatch a batch of jobs, you should use the `batch` method of the `Bus` facade. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the `then`, `catch`, and `finally` methods to define completion callbacks for the batch. Each of these callbacks will receive an `Illuminate\Bus\Batch` instance when they are invoked. In this example, we will imagine we are queueing a batch of jobs that each process a given number of rows from a CSV file:

```

use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->before(function (Batch $batch) {
    // The batch has been created but no jobs have been added...
})->progress(function (Batch $batch) {
    // A single job has completed successfully...
})->then(function (Batch $batch) {
    // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
    // First batch job failure detected...
})->finally(function (Batch $batch) {
    // The batch has finished executing...
})->dispatch();

return $batch->id;

```

The batch's ID, which may be accessed via the `$batch->id` property, may be used to [query the Laravel command bus](#) for information about the batch after it has been dispatched.



Since batch callbacks are serialized and executed at a later time by the Laravel queue, you should not use the `$this` variable within the callbacks.

Naming Batches

Some tools such as Laravel Horizon and Laravel Telescope may provide more user-friendly debug information for batches if batches are named. To assign an arbitrary name to a batch, you may call the `name` method while defining the batch:

```

$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import CSV')->dispatch();

```

Batch Connection and Queue

If you would like to specify the connection and queue that should be used for the batched jobs, you may use the `onConnection` and `onQueue` methods. All batched jobs must execute within the same connection and queue:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

Chains and Batches

You may define a set of [chained jobs](#) within a batch by placing the chained jobs within an array. For example, we may execute two job chains in parallel and execute a callback when both job chains have finished processing:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

Conversely, you may run batches of jobs within a [chain](#) by defining batches within the chain. For example, you could first run a batch of jobs to release multiple podcasts then a batch of jobs to send the release notifications:

```
use App\Jobs\FlushPodcastCache;
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new FlushPodcastCache,
    Bus::batch([
        new ReleasePodcast(1),
        new ReleasePodcast(2),
    ]),
    Bus::batch([
        new SendPodcastReleaseNotification(1),
        new SendPodcastReleaseNotification(2),
    ]),
])->dispatch();
```

Adding Jobs to Batches

Sometimes it may be useful to add additional jobs to a batch from within a batched job. This pattern can be useful when you need to batch thousands of jobs which may take too long to dispatch during a web request. So, instead, you may wish to dispatch an initial batch of "loader" jobs that hydrate the batch with even more jobs:

```
$batch = Bus::batch([
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import Contacts')->dispatch();
```

In this example, we will use the `LoadImportBatch` job to hydrate the batch with additional jobs. To accomplish this, we may use the `add` method on the batch instance that may be accessed via the job's `batch` method:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * Execute the job.
 */
public function handle(): void
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```



You may only add jobs to a batch from within a job that belongs to the same batch.

Inspecting Batches

The `Illuminate\Bus\Batch` instance that is provided to batch completion callbacks has a variety of properties and methods to assist you in interacting with and inspecting a given batch of jobs:

```
// The UUID of the batch...
$batch->id;

// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;

// The number of jobs that have failed...
$batch->failedJobs;

// The number of jobs that have been processed thus far...
$batch->processedJobs();

// The completion percentage of the batch (0-100)...
$batch->progress();

// Indicates if the batch has finished executing...
$batch->finished();

// Cancel the execution of the batch...
$batch->cancel();

// Indicates if the batch has been cancelled...
$batch->cancelled();
```

Returning Batches From Routes

All `Illuminate\Bus\Batch` instances are JSON serializable, meaning you can return them directly from one of your application's routes to retrieve a JSON payload containing information about the batch, including its completion progress. This makes it convenient to display information about the batch's completion progress in your application's UI.

To retrieve a batch by its ID, you may use the `Bus` facade's `findBatch` method:

```
use Illuminate\Support\Facades\Bus;
use Illuminate\Support\Facades\Route;

Route::get('/batch/{batchId}', function (string $batchId) {
    return Bus::findBatch($batchId);
});
```

Cancelling Batches

Sometimes you may need to cancel a given batch's execution. This can be accomplished by calling the `cancel` method on the `Illuminate\Bus\Batch` instance:

```
/**
 * Execute the job.
 */
public function handle(): void
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

    if ($this->batch()->cancelled()) {
        return;
    }
}
```

As you may have noticed in the previous examples, batched jobs should typically determine if their corresponding batch has been cancelled before continuing execution. However, for convenience, you may assign the [SkipIfBatchCancelled middleware](#) to the job instead. As its name indicates, this middleware will instruct Laravel to not process the job if its corresponding batch has been cancelled:

```
use Illuminate\Queue\Middleware\SkipIfBatchCancelled;

/**
 * Get the middleware the job should pass through.
 */
public function middleware(): array
{
    return [new SkipIfBatchCancelled];
}
```

Batch Failures

When a batched job fails, the `catch` callback (if assigned) will be invoked. This callback is only invoked for the first job that fails within the batch.

Allowing Failures

When a job within a batch fails, Laravel will automatically mark the batch as "cancelled". If you wish, you may disable this behavior so that a job failure does not automatically mark the batch as cancelled. This may be accomplished by calling the `allowFailures` method while dispatching the batch:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->allowFailures()->dispatch();
```

Retrying Failed Batch Jobs

For convenience, Laravel provides a `queue:retry-batch` Artisan command that allows you to easily retry all of the failed jobs for a given batch. The `queue:retry-batch` command accepts the UUID of the batch whose failed jobs should be retried:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

Pruning Batches

Without pruning, the `job_batches` table can accumulate records very quickly. To mitigate this, you should [schedule](#) the `queue:prune-batches` Artisan command to run daily:

```
$schedule->command('queue:prune-batches')->daily();
```

By default, all finished batches that are more than 24 hours old will be pruned. You may use the `hours` option when calling the command to determine how long to retain batch data. For example, the following command will delete all batches that finished over 48 hours ago:

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

Sometimes, your `jobs_batches` table may accumulate batch records for batches that never completed successfully, such as batches where a job failed and that job was never retried successfully. You may instruct the `queue:prune-batches` command to prune these unfinished batch records using the `unfinished` option:

```
$schedule->command('queue:prune-batches --hours=48 --unfinished=72')->daily();
```

Likewise, your `jobs_batches` table may also accumulate batch records for cancelled batches. You may instruct the `queue:prune-batches` command to prune these cancelled batch records using the `cancelled` option:

```
$schedule->command('queue:prune-batches --hours=48 --cancelled=72')->daily();
```

Storing Batches in DynamoDB

Laravel also provides support for storing batch meta information in [DynamoDB](#) instead of a relational database. However, you will need to manually create a DynamoDB table to store all of the batch records.

Typically, this table should be named `job_batches`, but you should name the table based on the value of the `queue.batching.table` configuration value within your application's `queue` configuration file.

DynamoDB Batch Table Configuration

The `job_batches` table should have a string primary partition key named `application` and a string primary sort key named `id`. The `application` portion of the key will contain your application's name as defined by the `name` configuration value within your application's `app` configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store job batches for multiple Laravel applications.

In addition, you may define `ttl` attribute for your table if you would like to take advantage of [automatic batch pruning](#).

DynamoDB Configuration

Next, install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Then, set the `queue.batching.driver` configuration option's value to `dynamodb`. In addition, you should define `key`, `secret`, and `region` configuration options within the `batching` configuration array. These options will be used to authenticate with AWS. When using the `dynamodb` driver, the `queue.batching.database` configuration option is unnecessary:

```
'batching' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'job_batches',
],
]
```

Pruning Batches in DynamoDB

When utilizing [DynamoDB](#) to store job batch information, the typical pruning commands used to prune batches stored in a relational database will not work. Instead, you may utilize [DynamoDB's native TTL functionality](#) to automatically remove records for old batches.

If you defined your DynamoDB table with a `ttl` attribute, you may define configuration parameters to instruct Laravel how to prune batch records. The `queue.batching.ttl_attribute` configuration value defines the name of the attribute holding the TTL, while the `queue.batching.ttl` configuration value defines the number of seconds after which a batch record can be removed from the DynamoDB table, relative to the last time the record was updated:

```
'batching' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'job_batches',
    'ttl_attribute' => 'ttl',
    'ttl' => 60 * 60 * 24 * 7, // 7 days...
],
]
```

Queueing Closures

Instead of dispatching a job class to the queue, you may also dispatch a closure. This is great for quick, simple tasks that need to be executed outside of the current request cycle. When dispatching closures to the queue, the closure's code content is cryptographically signed so that it can not be modified in transit:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

Using the `catch` method, you may provide a closure that should be executed if the queued closure fails to complete successfully after exhausting all of your queue's [configured retry attempts](#):

```
use Throwable;

dispatch(function () use ($podcast) {
    $podcast->publish();
})->catch(function (Throwable $e) {
    // This job has failed...
});
```



Since `catch` callbacks are serialized and executed at a later time by the Laravel queue, you should not use the `$this` variable within `catch` callbacks.

Running the Queue Worker

The `queue:work` Command

Laravel includes an Artisan command that will start a queue worker and process new jobs as they are pushed onto the queue. You may run the worker using the `queue:work` Artisan command. Note that once the `queue:work` command has started, it will continue to run until it is manually stopped or you close your terminal:

```
php artisan queue:work
```



To keep the `queue:work` process running permanently in the background, you should use a process monitor such as [Supervisor](#) to ensure that the queue worker does not stop running.

You may include the `-v` flag when invoking the `queue:work` command if you would like the processed job IDs to be included in the command's output:

```
php artisan queue:work -v
```

Remember, queue workers are long-lived processes and store the booted application state in memory. As a result, they will not notice changes in your code base after they have been started. So, during your deployment process, be sure to [restart your queue workers](#). In addition, remember that any static state created or modified by your application will not be automatically reset between jobs.

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker when you want to reload your updated code or reset the application state; however, this command is significantly less efficient than the `queue:work` command:

```
php artisan queue:listen
```

Running Multiple Queue Workers

To assign multiple workers to a queue and process jobs concurrently, you should simply start multiple `queue:work` processes. This can either be done locally via multiple tabs in your terminal or in production using your process manager's configuration settings. [When using Supervisor](#), you may use the `numprocs` configuration value.

Specifying the Connection and Queue

You may also specify which queue connection the worker should utilize. The connection name passed to the `work` command should correspond to one of the connections defined in your `config/queue.php` configuration file:

```
php artisan queue:work redis
```

By default, the `queue:work` command only processes jobs for the default queue on a given connection. However, you may customize your queue worker even further by only processing particular queues for a given connection. For example,

if all of your emails are processed in an `emails` queue on your `redis` queue connection, you may issue the following command to start a worker that only processes that queue:

```
php artisan queue:work redis --queue=emails
```

Processing a Specified Number of Jobs

The `--once` option may be used to instruct the worker to only process a single job from the queue:

```
php artisan queue:work --once
```

The `--max-jobs` option may be used to instruct the worker to process the given number of jobs and then exit. This option may be useful when combined with [Supervisor](#) so that your workers are automatically restarted after processing a given number of jobs, releasing any memory they may have accumulated:

```
php artisan queue:work --max-jobs=1000
```

Processing All Queued Jobs and Then Exiting

The `--stop-when-empty` option may be used to instruct the worker to process all jobs and then exit gracefully. This option can be useful when processing Laravel queues within a Docker container if you wish to shutdown the container after the queue is empty:

```
php artisan queue:work --stop-when-empty
```

Processing Jobs for a Given Number of Seconds

The `--max-time` option may be used to instruct the worker to process jobs for the given number of seconds and then exit. This option may be useful when combined with [Supervisor](#) so that your workers are automatically restarted after processing jobs for a given amount of time, releasing any memory they may have accumulated:

```
# Process jobs for one hour and then exit...  
php artisan queue:work --max-time=3600
```

Worker Sleep Duration

When jobs are available on the queue, the worker will keep processing jobs with no delay in between jobs. However, the `sleep` option determines how many seconds the worker will "sleep" if there are no jobs available. Of course, while sleeping, the worker will not process any new jobs:

```
php artisan queue:work --sleep=3
```

Maintenance Mode and Queues

While your application is in [maintenance mode](#), no queued jobs will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

To force your queue workers to process jobs even if maintenance mode is enabled, you may use `--force` option:

```
php artisan queue:work --force
```

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should release any heavy resources after each job completes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done processing the image.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` configuration file, you may set the default `queue` for your `redis` connection to `low`. However, occasionally you may wish to push a job to a `high` priority queue like so:

```
dispatch((new Job)->onQueue('high'));
```

To start a worker that verifies that all of the `high` queue jobs are processed before continuing to any jobs on the `low` queue, pass a comma-delimited list of queue names to the `work` command:

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

Queue Workers and Deployment

Since queue workers are long-lived processes, they will not notice changes to your code without being restarted. So, the simplest way to deploy an application using queue workers is to restart the workers during your deployment process. You may gracefully restart all of the workers by issuing the `queue:restart` command:

```
php artisan queue:restart
```

This command will instruct all queue workers to gracefully exit after they finish processing their current job so that no existing jobs are lost. Since the queue workers will exit when the `queue:restart` command is executed, you should be running a process manager such as [Supervisor](#) to automatically restart the queue workers.



The queue uses the `cache` to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Job Expirations and Timeouts

Job Expiration

In your `config/queue.php` configuration file, each queue connection defines a `retry_after` option. This option specifies how many seconds the queue connection should wait before retrying a job that is being processed. For example, if the value of `retry_after` is set to `90`, the job will be released back onto the queue if it has been processing for 90 seconds without being released or deleted. Typically, you should set the `retry_after` value to the maximum number of seconds your jobs should reasonably take to complete processing.



The only queue connection which does not contain a `retry_after` value is Amazon SQS. SQS will retry the job based on the [Default Visibility Timeout](#) which is managed within the AWS console.

Worker Timeouts

The `queue:work` Artisan command exposes a `--timeout` option. By default, the `--timeout` value is 60 seconds. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will

exit with an error. Typically, the worker will be restarted automatically by a [process manager configured on your server](#):

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

The `retry_after` configuration option and the `--timeout` CLI option are different, but work together to ensure that jobs are not lost and that jobs are only successfully processed once.



The `--timeout` value should always be at least several seconds shorter than your `retry_after` configuration value. This will ensure that a worker processing a frozen job is always terminated before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

Supervisor Configuration

In production, you need a way to keep your `queue:work` processes running. A `queue:work` process may stop running for a variety of reasons, such as an exceeded worker timeout or the execution of the `queue:restart` command.

For this reason, you need to configure a process monitor that can detect when your `queue:work` processes exit and automatically restart them. In addition, process monitors can allow you to specify how many `queue:work` processes you would like to run concurrently. Supervisor is a process monitor commonly used in Linux environments and we will discuss how to configure it in the following documentation.

Installing Supervisor

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

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



If configuring and managing Supervisor yourself sounds overwhelming, consider using [Laravel Forge](#), which will automatically install and configure Supervisor for your production Laravel projects.

Configuring Supervisor

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

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

In this example, the `numprocs` directive will instruct Supervisor to run eight `queue:work` processes and monitor all of them, automatically restarting them if they fail. You should change the `command` directive of the configuration to reflect your desired queue connection and worker options.



You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start "laravel-worker:*
```

For more information on Supervisor, consult the [Supervisor documentation](#).

Dealing With Failed Jobs

Sometimes your queued jobs will fail. Don't worry, things don't always go as planned! Laravel includes a convenient way to specify the maximum number of times a job should be attempted. After an asynchronous job has exceeded this number of attempts, it will be inserted into the `failed_jobs` database table. Synchronously dispatched jobs that fail are not stored in this table and their exceptions are immediately handled by the application.

A migration to create the `failed_jobs` table is typically already present in new Laravel applications. However, if your application does not contain a migration for this table, you may use the `queue:failed-table` command to create the migration:

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

When running a `queue worker` process, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:work` command. If you do not specify a value for the `--tries` option, jobs will only

be attempted once or as many times as specified by the job class' `$tries` property:

```
php artisan queue:work redis --tries=3
```

Using the `--backoff` option, you may specify how many seconds Laravel should wait before retrying a job that has encountered an exception. By default, a job is immediately released back onto the queue so that it may be attempted again:

```
php artisan queue:work redis --tries=3 --backoff=3
```

If you would like to configure how many seconds Laravel should wait before retrying a job that has encountered an exception on a per-job basis, you may do so by defining a `backoff` property on your job class:

```
/**  
 * The number of seconds to wait before retrying the job.  
 *  
 * @var int  
 */  
public $backoff = 3;
```

If you require more complex logic for determining the job's backoff time, you may define a `backoff` method on your job class:

```
/**  
 * Calculate the number of seconds to wait before retrying the job.  
 */  
public function backoff(): int  
{  
    return 3;  
}
```

You may easily configure "exponential" backoffs by returning an array of backoff values from the `backoff` method. In this example, the retry delay will be 1 second for the first retry, 5 seconds for the second retry, 10 seconds for the third retry, and 10 seconds for every subsequent retry if there are more attempts remaining:

```
/**  
 * Calculate the number of seconds to wait before retrying the job.  
 *  
 * @return array<int, int>  
 */  
public function backoff(): array  
{  
    return [1, 5, 10];  
}
```

Cleaning Up After Failed Jobs

When a particular job fails, you may want to send an alert to your users or revert any actions that were partially completed by the job. To accomplish this, you may define a `failed` method on your job class. The `Throwable` instance that caused the job to fail will be passed to the `failed` method:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * Execute the job.
     */
    public function handle(AudioProcessor $processor): void
    {
        // Process uploaded podcast...
    }

    /**
     * Handle a job failure.
     */
    public function failed(?Throwable $exception): void
    {
        // Send user notification of failure, etc...
    }
}
```



A new instance of the job is instantiated before invoking the `failed` method; therefore, any class property modifications that may have occurred within the `handle` method will be lost.

Retrying Failed Jobs

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

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, failure time, and other information about the job. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of `ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece`, issue the following command:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
```

If necessary, you may pass multiple IDs to the command:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece 91401d2c-0784-4f43-824c-34f94a33c24d
```

You may also retry all of the failed jobs for a particular queue:

```
php artisan queue:retry --queue=name
```

To retry all of your failed jobs, execute the `queue:retry` command and pass `all` as the ID:

```
php artisan queue:retry all
```

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

```
php artisan queue:forget 91401d2c-0784-4f43-824c-34f94a33c24d
```



When using [Horizon](#), you should use the `horizon:forget` command to delete a failed job instead of the `queue:forget` command.

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

```
php artisan queue:flush
```

Ignoring Missing Models

When injecting an Eloquent model into a job, the model is automatically serialized before being placed on the queue and re-retrieved from the database when the job is processed. However, if the model has been deleted while the job was waiting to be processed by a worker, your job may fail with a `ModelNotFoundException`.

For convenience, you may choose to automatically delete jobs with missing models by setting your job's `deleteWhenMissingModels` property to `true`. When this property is set to `true`, Laravel will quietly discard the job without raising an exception:

```
/**
 * Delete the job if its models no longer exist.
 *
 * @var bool
 */
public $deleteWhenMissingModels = true;
```

Pruning Failed Jobs

You may prune the records in your application's `failed_jobs` table by invoking the `queue:prune-failed` Artisan command:

```
php artisan queue:prune-failed
```

By default, all the failed job records that are more than 24 hours old will be pruned. If you provide the `--hours` option to the command, only the failed job records that were inserted within the last N number of hours will be retained. For example, the following command will delete all the failed job records that were inserted more than 48 hours ago:

```
php artisan queue:prune-failed --hours=48
```

Storing Failed Jobs in DynamoDB

Laravel also provides support for storing your failed job records in [DynamoDB](#) instead of a relational database table. However, you must manually create a DynamoDB table to store all of the failed job records. Typically, this table should be named `failed_jobs`, but you should name the table based on the value of the `queue.failed.table` configuration value within your application's `queue` configuration file.

The `failed_jobs` table should have a string primary partition key named `application` and a string primary sort key named `uuid`. The `application` portion of the key will contain your application's name as defined by the `name` configuration value within your application's `app` configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store failed jobs for multiple Laravel applications.

In addition, ensure that you install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Next, set the `queue.failed.driver` configuration option's value to `dynamodb`. In addition, you should define `key`, `secret`, and `region` configuration options within the failed job configuration array. These options will be used to authenticate with AWS. When using the `dynamodb` driver, the `queue.failed.database` configuration option is unnecessary:

```
'failed' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'failed_jobs',
],
```

Disabling Failed Job Storage

You may instruct Laravel to discard failed jobs without storing them by setting the `queue.failed.driver` configuration option's value to `null`. Typically, this may be accomplished via the `QUEUE_FAILED_DRIVER` environment variable:

```
QUEUE_FAILED_DRIVER=null
```

Failed Job Events

If you would like to register an event listener that will be invoked when a job fails, you may use the `Queue` facade's `failing` method. For example, we may attach a closure to this event from the `boot` method of the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
}
```

Clearing Jobs From Queues



When using `Horizon`, you should use the `horizon:clear` command to clear jobs from the queue instead of the `queue:clear` command.

If you would like to delete all jobs from the default queue of the default connection, you may do so using the `queue:clear` Artisan command:

```
php artisan queue:clear
```

You may also provide the `connection` argument and `queue` option to delete jobs from a specific connection and queue:

```
php artisan queue:clear redis --queue=emails
```



Clearing jobs from queues is only available for the SQS, Redis, and database queue drivers. In addition, the SQS message deletion process takes up to 60 seconds, so jobs sent to the SQS queue up to 60 seconds after you clear the queue might also be deleted.

Monitoring Your Queues

If your queue receives a sudden influx of jobs, it could become overwhelmed, leading to a long wait time for jobs to complete. If you wish, Laravel can alert you when your queue job count exceeds a specified threshold.

To get started, you should schedule the `queue:monitor` command to run every minute. The command accepts the names of the queues you wish to monitor as well as your desired job count threshold:

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

Scheduling this command alone is not enough to trigger a notification alerting you of the queue's overwhelmed status.

When the command encounters a queue that has a job count exceeding your threshold, an

`Illuminate\Queue\Events\QueueBusy` event will be dispatched. You may listen for this event within your application's `EventServiceProvider` in order to send a notification to you or your development team:

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * Register any other events for your application.
 */
public function boot(): void
{
    Event::listen(function (QueueBusy $event) {
        Notification::route('mail', 'dev@example.com')
            ->notify(new QueueHasLongWaitTime(
                $event->connection,
                $event->queue,
                $event->size
            ));
    });
}
```

Testing

When testing code that dispatches jobs, you may wish to instruct Laravel to not actually execute the job itself, since the job's code can be tested directly and separately of the code that dispatches it. Of course, to test the job itself, you may instantiate a job instance and invoke the `handle` method directly in your test.

You may use the `Queue` facade's `fake` method to prevent queued jobs from actually being pushed to the queue. After calling the `Queue` facade's `fake` method, you may then assert that the application attempted to push jobs to the queue:

```
<?php

namespace Tests\Feature;

use App\Jobs\AnotherJob;
use App\Jobs\FinalJob;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Queue::fake();

        // Perform order shipping...

        // Assert that no jobs were pushed...
        Queue::assertNothingPushed();

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was pushed twice...
        Queue::assertPushed(ShipOrder::class, 2);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::class);

        // Assert that a Closure was pushed to the queue...
        Queue::assertClosurePushed();

        // Assert the total number of jobs that were pushed...
        Queue::assertCount(3);
    }
}
```

You may pass a closure to the `assertPushed` or `assertNotPushed` methods in order to assert that a job was pushed that passes a given "truth test". If at least one job was pushed that passes the given truth test then the assertion will be successful:

```
Queue::assertPushed(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

Faking a Subset of Jobs

If you only need to fake specific jobs while allowing your other jobs to execute normally, you may pass the class names of the jobs that should be faked to the `fake` method:

```
public function test_orders_can_be_shipped(): void
{
    Queue::fake([
        ShipOrder::class,
    ]);

    // Perform order shipping...

    // Assert a job was pushed twice...
    Queue::assertPushed(ShipOrder::class, 2);
}
```

You may fake all jobs except for a set of specified jobs using the `except` method:

```
Queue::fake()->except([
    ShipOrder::class,
]);
```

Testing Job Chains

To test job chains, you will need to utilize the `Bus` facade's faking capabilities. The `Bus` facade's `assertChained` method may be used to assert that a chain of jobs was dispatched. The `assertChained` method accepts an array of chained jobs as its first argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;

Bus::fake();

// ...

Bus::assertChained([
    ShipOrder::class,
    RecordShipment::class,
    UpdateInventory::class
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Bus::assertChained([
    new ShipOrder,
    new RecordShipment,
    new UpdateInventory,
]);
```

You may use the `assertDispatchedWithoutChain` method to assert that a job was pushed without a chain of jobs:

```
Bus::assertDispatchedWithoutChain(ShipOrder::class);
```

Testing Chained Batches

If your job chain `contains a batch of jobs`, you may assert that the chained batch matches your expectations by inserting a `Bus::chainedBatch` definition within your chain assertion:

```
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::assertChained([
    new ShipOrder,
    Bus::chainedBatch(function (PendingBatch $batch) {
        return $batch->jobs->count() === 3;
    }),
    new UpdateInventory,
]);
```

Testing Job Batches

The `Bus` facade's `assertBatched` method may be used to assert that a `batch of jobs` was dispatched. The closure given to the `assertBatched` method receives an instance of `Illuminate\Bus\PendingBatch`, which may be used to inspect the jobs within the batch:

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::fake();

// ...

Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
        $batch->jobs->count() === 10;
});
```

You may use the `assertBatchCount` method to assert that a given number of batches were dispatched:

```
Bus::assertBatchCount(3);
```

You may use `assertNothingBatched` to assert that no batches were dispatched:

```
Bus::assertNothingBatched();
```

Testing Job / Batch Interaction

In addition, you may occasionally need to test an individual job's interaction with its underlying batch. For example, you may need to test if a job cancelled further processing for its batch. To accomplish this, you need to assign a fake batch to the job via the `withFakeBatch` method. The `withFakeBatch` method returns a tuple containing the job instance and the fake batch:

```

[$job, $batch] = (new ShipOrder)->withFakeBatch();

$job->handle();

$this->assertTrue($batch->cancelled());
$this->assertEmpty($batch->added);

```

Job Events

Using the `before` and `after` methods on the `Queue facade`, you may specify callbacks to be executed before or after a queued job is processed. These callbacks are a great opportunity to perform additional logging or increment statistics for a dashboard. Typically, you should call these methods from the `boot` method of a [service provider](#). For example, we may use the `AppServiceProvider` that is included with Laravel:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }
}

```

Using the `looping` method on the `Queue facade`, you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a closure to rollback any transactions that were left open by a

previously failed job:

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Queue;

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

Rate Limiting

- [Introduction](#)
 - [Cache Configuration](#)
- [Basic Usage](#)
 - [Manually Incrementing Attempts](#)
 - [Clearing Attempts](#)

Introduction

Laravel includes a simple to use rate limiting abstraction which, in conjunction with your application's [cache](#), provides an easy way to limit any action during a specified window of time.



If you are interested in rate limiting incoming HTTP requests, please consult the [rate limiter middleware documentation](#).

Cache Configuration

Typically, the rate limiter utilizes your default application cache as defined by the `default` key within your application's `cache` configuration file. However, you may specify which cache driver the rate limiter should use by defining a `limiter` key within your application's `cache` configuration file:

```
'default' => 'memcached',  
  
'limiter' => 'redis',
```

Basic Usage

The `Illuminate\Support\Facades\RateLimiter` facade may be used to interact with the rate limiter. The simplest method offered by the rate limiter is the `attempt` method, which limits a given callback for a given number of seconds.

The `attempt` method returns `false` when the callback has no remaining attempts available; otherwise, the `attempt` method will return the callback's result or `true`. The first argument accepted by the `attempt` method is a rate limiter "key", which may be any string of your choosing that represents the action being rate limited:

```
use Illuminate\Support\Facades\RateLimiter;  
  
$executed = RateLimiter::attempt(  
    'send-message:'.$user->id,  
    $perMinute = 5,  
    function() {  
        // Send message...  
    }  
);  
  
if (! $executed) {  
    return 'Too many messages sent!';  
}
```

If necessary, you may provide a fourth argument to the `attempt` method, which is the "decay rate", or the number of seconds until the available attempts are reset. For example, we can modify the example above to allow five attempts every two minutes:

```
$executed = RateLimiter::attempt(
    'send-message:'.$user->id,
    $perTwoMinutes = 5,
    function() {
        // Send message...
    },
    $decayRate = 120,
);
```

Manually Incrementing Attempts

If you would like to manually interact with the rate limiter, a variety of other methods are available. For example, you may invoke the `tooManyAttempts` method to determine if a given rate limiter key has exceeded its maximum number of allowed attempts per minute:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
    return 'Too many attempts!';
}

RateLimiter::increment('send-message:'.$user->id);

// Send message...
```

Alternatively, you may use the `remaining` method to retrieve the number of attempts remaining for a given key. If a given key has retries remaining, you may invoke the `increment` method to increment the number of total attempts:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::remaining('send-message:'.$user->id, $perMinute = 5)) {
    RateLimiter::increment('send-message:'.$user->id);

    // Send message...
}
```

If you would like to increment the value for a given rate limiter key by more than one, you may provide the desired amount to the `increment` method:

```
RateLimiter::increment('send-message:'.$user->id, amount: 5);
```

Determining Limiter Availability

When a key has no more attempts left, the `availableIn` method returns the number of seconds remaining until more attempts will be available:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
    $seconds = RateLimiter::availableIn('send-message:'.$user->id);

    return 'You may try again in '.$seconds.' seconds.';
}

RateLimiter::increment('send-message:'.$user->id);

// Send message...
```

Clearing Attempts

You may reset the number of attempts for a given rate limiter key using the `clear` method. For example, you may reset the number of attempts when a given message is read by the receiver:

```
use App\Models\Message;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Mark the message as read.
 */
public function read(Message $message): Message
{
    $message->markAsRead();

    RateLimiter::clear('send-message:'.$message->user_id);

    return $message;
}
```

Strings

- [Introduction](#)
- [Available Methods](#)

Introduction

Laravel includes a variety of functions for manipulating string values. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

Available Methods

Strings

—	Str::kebab	Str::slug
class_basename	Str::lcfirst	Str::snake
e	Str::length	Str::squish
preg_replace_array	Str::limit	Str::start
Str::after	Str::lower	Str::startsWith
Str::afterLast	Str::markdown	Str::studly
Str::apa	Str::mask	Str::substr
Str::ascii	Str::orderedUuid	Str::substrCount
Str::before	Str::padBoth	Str::substrReplace
Str::beforeLast	Str::padLeft	Str::swap
Str::between	Str::padRight	Str::take
Str::betweenFirst	Str::password	Str::title
Str::camel	Str::plural	Str::toBase64
Str::charAt	Str::pluralStudy	Str::toHtmlString
Str::contains	Str::position	Str::ucfirst
Str::containsAll	Str::random	Str::ucsplit
Str::endsWith	Str::remove	Str::upper
Str::excerpt	Str::repeat	Str::ulid
Str::finish	Str::replace	Str::unwrap
Str::headline	Str::replaceArray	Str::uuid
Str::inlineMarkdown	Str::replaceFirst	Str::wordCount
Str::is	Str::replaceLast	Str::wordWrap
Str::isAscii	Str::replaceMatches	Str::words
Str::isJson	Str::replaceStart	Str::wrap
Str::isUlid	Str::replaceEnd	str
Str::isUrl	Str::reverse	trans
Str::isUuid	Str::singular	trans_choice

Fluent Strings

after	camel	finish
afterLast	charAt	headline
apa	classBasename	inlineMarkdown
append	contains	is
ascii	containsAll	isAscii
basename	dirname	isEmpty
before	endsWith	isNotEmpty
beforeLast	excerpt	isJson
between	exactly	isUlid
betweenFirst	explode	isUrl

<u>isUuid</u>	<u>replaceArray</u>	<u>title</u>
<u>kebab</u>	<u>replaceFirst</u>	<u>toBase64</u>
<u>lcfirst</u>	<u>replaceLast</u>	<u>trim</u>
<u>length</u>	<u>replaceMatches</u>	<u>ucfirst</u>
<u>limit</u>	<u>replaceStart</u>	<u>ucsplit</u>
<u>lower</u>	<u>replaceEnd</u>	<u>unwrap</u>
<u>ltrim</u>	<u>rtrim</u>	<u>upper</u>
<u>markdown</u>	<u>scan</u>	<u>when</u>
<u>mask</u>	<u>singular</u>	<u>whenContains</u>
<u>match</u>	<u>slug</u>	<u>whenContainsAll</u>
<u>matchAll</u>	<u>snake</u>	<u>whenEmpty</u>
<u>isMatch</u>	<u>split</u>	<u>whenNotEmpty</u>
<u>newLine</u>	<u>squish</u>	<u>whenStartsWith</u>
<u>padBoth</u>	<u>start</u>	<u>whenEndsWith</u>
<u>padLeft</u>	<u>startsWith</u>	<u>whenExactly</u>
<u>padRight</u>	<u>stripTags</u>	<u>whenNotExactly</u>
<u>pipe</u>	<u>studly</u>	<u>whenIs</u>
<u>plural</u>	<u>substr</u>	<u>whenIsAscii</u>
<u>position</u>	<u>substrReplace</u>	<u>whenIsUlid</u>
<u>prepend</u>	<u>swap</u>	<u>whenIsUuid</u>
<u>remove</u>	<u>take</u>	<u>whenTest</u>
<u>repeat</u>	<u>tap</u>	<u>wordCount</u>
<u>replace</u>	<u>test</u>	<u>words</u>

Strings

__()

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

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

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

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

class_basename()

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

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

// Baz
```

e()

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

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

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

preg_replace_array()

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

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

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

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

Str::after()

The `Str::after()` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

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

// ' my name'
```

Str::afterLast()

The `Str::afterLast()` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::afterLast('App\Http\Controllers\Controller', '\\');

// 'Controller'
```

Str::apa()

The `Str::apa()` method converts the given string to title case following the [APA guidelines](#):

```
use Illuminate\Support\Str;

$title = Str::apa('Creating A Project');

// 'Creating a Project'
```

Str::ascii()

The `Str::ascii()` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$slice = Str::ascii('û');

// 'u'
```

Str::before()

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

```
use Illuminate\Support\Str;

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

// 'This is '
```

Str::beforeLast()

The `Str::beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

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

// 'This '
```

Str::between()

The `Str::between` method returns the portion of a string between two values:

```
use Illuminate\Support\Str;

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

// ' is my '
```

Str::betweenFirst()

The `Str::betweenFirst` method returns the smallest possible portion of a string between two values:

```
use Illuminate\Support\Str;

$slice = Str::betweenFirst('[a] bc [d]', '[', ']');

// 'a'
```

Str::camel()

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

```
use Illuminate\Support\Str;

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

// 'fooBar'
```

Str::charAt()

The `Str::charAt` method returns the character at the specified index. If the index is out of bounds, `false` is returned:

```
use Illuminate\Support\Str;

$character = Str::charAt('This is my name.', 6);

// 's'
```

Str::contains()

The `Str::contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

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

// true
```

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

```
use Illuminate\Support\Str;

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

// true
```

Str::containsAll()

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

```
use Illuminate\Support\Str;

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

// true
```

Str::endsWith()

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

```
use Illuminate\Support\Str;

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

// true
```

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

```
use Illuminate\Support\Str;

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

// true

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

// false
```

Str::excerpt()

The `Str::excerpt` method extracts an excerpt from a given string that matches the first instance of a phrase within that string:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'my', [
    'radius' => 3
]);

// '...is my na...'
```

The `radius` option, which defaults to `100`, allows you to define the number of characters that should appear on each side of the truncated string.

In addition, you may use the `omission` option to define the string that will be prepended and appended to the truncated string:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'name', [
    'radius' => 3,
    'omission' => '(...)'
]);

// '(...) my name'
```

Str::finish()

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

```
use Illuminate\Support\Str;

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

// this/string/

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

// this/string/
```

Str::headline()

The `Str::headline` method will convert strings delimited by casing, hyphens, or underscores into a space delimited string with each word's first letter capitalized:

```
use Illuminate\Support\Str;

$headline = Str::headline('steve_jobs');

// Steve Jobs

$headline = Str::headline('EmailNotificationSent');

// Email Notification Sent
```

`Str::inlineMarkdown()`

The `Str::inlineMarkdown` method converts GitHub flavored Markdown into inline HTML using [CommonMark](#). However, unlike the `markdown` method, it does not wrap all generated HTML in a block-level element:

```
use Illuminate\Support\Str;

$html = Str::inlineMarkdown('**Laravel**');

// <strong>Laravel</strong>
```

Markdown Security

By default, Markdown supports raw HTML, which will expose Cross-Site Scripting (XSS) vulnerabilities when used with raw user input. As per the [CommonMark Security documentation](#), you may use the `html_input` option to either escape or strip raw HTML, and the `allow_unsafe_links` option to specify whether to allow unsafe links. If you need to allow some raw HTML, you should pass your compiled Markdown through an HTML Purifier:

```
use Illuminate\Support\Str;

Str::inlineMarkdown('Inject: <script>alert("Hello XSS!");</script>', [
    'html_input' => 'strip',
    'allow_unsafe_links' => false,
]);

// Inject: alert("Hello XSS!");
```

`Str::is()`

The `Str::is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values:

```
use Illuminate\Support\Str;

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

// true

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

// false
```

Str::isAscii()

The `Str::isAscii` method determines if a given string is 7 bit ASCII:

```
use Illuminate\Support\Str;

$isAscii = Str::isAscii('Taylor');

// true

$isAscii = Str::isAscii('ü');

// false
```

Str::isJson()

The `Str::isJson` method determines if the given string is valid JSON:

```
use Illuminate\Support\Str;

$result = Str::isJson('[1,2,3]');

// true

$result = Str::isJson('{"first": "John", "last": "Doe"}');

// true

$result = Str::isJson('{first: "John", last: "Doe"}');

// false
```

Str::isUrl()

The `Str::isUrl` method determines if the given string is a valid URL:

```
use Illuminate\Support\Str;

$isUrl = Str::isUrl('http://example.com');

// true

$isUrl = Str::isUrl('laravel');

// false
```

The `isUrl` method considers a wide range of protocols as valid. However, you may specify the protocols that should be considered valid by providing them to the `isUrl` method:

```
$isUrl = Str::isUrl('http://example.com', ['http', 'https']);
```

Str::isUlid()

The `Str::isUlid` method determines if the given string is a valid ULID:

```
use Illuminate\Support\Str;

$isUlid = Str::isUlid('01gd6r360bp37zj17nxb55yv40');

// true

$isUlid = Str::isUlid('laravel');

// false
```

Str::isUuid()

The `Str::isUuid` method determines if the given string is a valid UUID:

```
use Illuminate\Support\Str;

$isUuid = Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');

// true

$isUuid = Str::isUuid('laravel');

// false
```

Str::kebab()

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

```
use Illuminate\Support\Str;

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

// foo-bar
```

Str::lcfirst()

The `Str::lcfirst` method returns the given string with the first character lowercased:

```
use Illuminate\Support\Str;

$string = Str::lcfirst('Foo Bar');

// foo Bar
```

Str::length()

The `Str::length` method returns the length of the given string:

```
use Illuminate\Support\Str;

$length = Str::length('Laravel');

// 7
```

Str::limit()

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

```
use Illuminate\Support\Str;

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

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

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

```
use Illuminate\Support\Str;

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

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

Str::lower()

The `Str::lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$converted = Str::lower('LARAVEL');

// laravel
```

Str::markdown()

The `Str::markdown` method converts GitHub flavored Markdown into HTML using [CommonMark](#):

```
use Illuminate\Support\Str;

$html = Str::markdown('# Laravel');

// <h1>Laravel</h1>

$html = Str::markdown('# Taylor <b>Otwell</b>', [
    'html_input' => 'strip',
]);

// <h1>Taylor Otwell</h1>
```

Markdown Security

By default, Markdown supports raw HTML, which will expose Cross-Site Scripting (XSS) vulnerabilities when used with raw user input. As per the [CommonMark Security documentation](#), you may use the `html_input` option to either escape or strip raw HTML, and the `allow_unsafe_links` option to specify whether to allow unsafe links. If you need to allow some raw HTML, you should pass your compiled Markdown through an HTML Purifier:

```
use Illuminate\Support\Str;

Str::markdown('Inject: <script>alert("Hello XSS!");</script>', [
    'html_input' => 'strip',
    'allow_unsafe_links' => false,
]);

// <p>Inject: alert("Hello XSS!");</p>
```

Str::mask()

The `Str::mask` method masks a portion of a string with a repeated character, and may be used to obfuscate segments of strings such as email addresses and phone numbers:

```
use Illuminate\Support\Str;

$string = Str::mask('taylor@example.com', '*', 3);

// tay*****@example.com
```

If needed, you provide a negative number as the third argument to the `mask` method, which will instruct the method to begin masking at the given distance from the end of the string:

```
$string = Str::mask('taylor@example.com', '*', -15, 3);

// tay***@example.com
```

Str::orderedUuid()

The `Str::orderedUuid` method generates a "timestamp first" UUID that may be efficiently stored in an indexed database column. Each UUID that is generated using this method will be sorted after UUIDs previously generated using the method:

```
use Illuminate\Support\Str;

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

Str::padBoth()

The `Str::padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padBoth('James', 10, '_');

// '__James__'

$padded = Str::padBoth('James', 10);

// ' James '
```

Str::padLeft()

The `Str::padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padLeft('James', 10, '-');

// '-----James'

$padded = Str::padLeft('James', 10);

// '      James'
```

`Str::padRight()`

The `Str::padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padRight('James', 10, '-');

// 'James-----'

$padded = Str::padRight('James', 10);

// 'James      '
```

`Str::password()`

The `Str::password` method may be used to generate a secure, random password of a given length. The password will consist of a combination of letters, numbers, symbols, and spaces. By default, passwords are 32 characters long:

```
use Illuminate\Support\Str;

$password = Str::password();

// 'EbJo2vE-AS:U,$%_gkrV4n,q~1xy/-_4'

$password = Str::password(12);

// 'qwuar>#V|i]N'
```

`Str::plural()`

The `Str::plural` method converts a singular word string to its plural form. This function supports [any of the languages support by Laravel's pluralizer](#):

```
use Illuminate\Support\Str;

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

// cars

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

// children
```

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

```
use Illuminate\Support\Str;

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

// children

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

// child
```

Str::pluralStudy()

The `Str::pluralStudy` method converts a singular word string formatted in study caps case to its plural form. This function supports [any of the languages support by Laravel's pluralizer](#):

```
use Illuminate\Support\Str;

$plural = Str::pluralStudy('VerifiedHuman');

// VerifiedHumans

$plural = Str::pluralStudy('UserFeedback');

// UserFeedback
```

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

```
use Illuminate\Support\Str;

$plural = Str::pluralStudy('VerifiedHuman', 2);

// VerifiedHumans

$singular = Str::pluralStudy('VerifiedHuman', 1);

// VerifiedHuman
```

Str::position()

The `Str::position` method returns the position of the first occurrence of a substring in a string. If the substring does not exist in the given string, `false` is returned:

```
use Illuminate\Support\Str;

$position = Str::position('Hello, World!', 'Hello');

// 0

$position = Str::position('Hello, World!', 'W');

// 7
```

Str::random()

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

```
use Illuminate\Support\Str;

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

During testing, it may be useful to "fake" the value that is returned by the `Str::random` method. To accomplish this, you may use the `createRandomStringsUsing` method:

```
Str::createRandomStringsUsing(function () {
    return 'fake-random-string';
});
```

To instruct the `random` method to return to generating random strings normally, you may invoke the `createRandomStringsNormally` method:

```
Str::createRandomStringsNormally();
```

Str::remove()

The `Str::remove` method removes the given value or array of values from the string:

```
use Illuminate\Support\Str;

$string = 'Peter Piper picked a peck of pickled peppers.';

$removed = Str::remove('e', $string);

// Ptr Pipr pickd a pck of pickld ppprs.
```

You may also pass `false` as a third argument to the `remove` method to ignore case when removing strings.

Str::repeat()

The `Str::repeat` method repeats the given string:

```
use Illuminate\Support\Str;

$string = 'a';

$repeat = Str::repeat($string, 5);

// aaaaa
```

Str::replace()

The `Str::replace` method replaces a given string within the string:

```
use Illuminate\Support\Str;

$string = 'Laravel 8.x';

$replaced = Str::replace('8.x', '9.x', $string);

// Laravel 9.x
```

The `replace` method also accepts a `caseSensitive` argument. By default, the `replace` method is case sensitive:

```
Str::replace('Framework', 'Laravel', caseSensitive: false);
```

Str::replaceArray()

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

```
use Illuminate\Support\Str;

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

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

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

Str::replaceFirst()

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

```
use Illuminate\Support\Str;

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

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

Str::replaceLast()

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

```
use Illuminate\Support\Str;

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

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

Str::replaceMatches()

The `Str::replaceMatches` method replaces all portions of a string matching a pattern with the given replacement string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceMatches(
    pattern: '/[^\w\d]+/',
    replace: '',
    subject: '(+1) 501-555-1000'
)

// '15015551000'
```

The `replaceMatches` method also accepts a closure that will be invoked with each portion of the string matching the given pattern, allowing you to perform the replacement logic within the closure and return the replaced value:

```
use Illuminate\Support\Str;

$replaced = Str::replaceMatches('/\d/', function (array $matches) {
    return '[' . $matches[0] . ']';
}, '123');

// '[1][2][3]'
```

Str::replaceStart()

The `Str::replaceStart` method replaces the first occurrence of the given value only if the value appears at the start of the string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceStart('Hello', 'Laravel', 'Hello World');

// Laravel World

$replaced = Str::replaceStart('World', 'Laravel', 'Hello World');

// Hello World
```

Str::replaceEnd()

The `Str::replaceEnd` method replaces the last occurrence of the given value only if the value appears at the end of the string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceEnd('World', 'Laravel', 'Hello World');

// Hello Laravel

$replaced = Str::replaceEnd('Hello', 'Laravel', 'Hello World');

// Hello World
```

Str::reverse()

The `Str::reverse` method reverses the given string:

```
use Illuminate\Support\Str;

$reversed = Str::reverse('Hello World');

// dlroW olleH
```

Str::singular()

The `Str::singular` method converts a string to its singular form. This function supports [any of the languages supported by Laravel's pluralizer](#):

```
use Illuminate\Support\Str;

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

// car

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

// child
```

Str::slug()

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

```
use Illuminate\Support\Str;

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

// laravel-5-framework
```

Str::snake()

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

```
use Illuminate\Support\Str;

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

// foo_bar

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

// foo-bar
```

Str::squish()

The `Str::squish` method removes all extraneous white space from a string, including extraneous white space between words:

```
use Illuminate\Support\Str;

$string = Str::squish('    laravel    framework    ');

// laravel framework
```

Str::start()

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

```
use Illuminate\Support\Str;

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

// /this/string

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

// /this/string
```

Str::startsWith()

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

```
use Illuminate\Support\Str;

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

// true
```

If an array of possible values is passed, the `startsWith` method will return `true` if the string begins with any of the given values:

```
$result = Str::startsWith('This is my name', ['This', 'That', 'There']);

// true
```

Str::studly()

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

```
use Illuminate\Support\Str;

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

// FooBar
```

`Str::substr()`

The `Str::substr` method returns the portion of string specified by the start and length parameters:

```
use Illuminate\Support\Str;

$converted = Str::substr('The Laravel Framework', 4, 7);

// Laravel
```

`Str::substrCount()`

The `Str::substrCount` method returns the number of occurrences of a given value in the given string:

```
use Illuminate\Support\Str;

$count = Str::substrCount('If you like ice cream, you will like snow cones.', 'like');

// 2
```

`Str::substrReplace()`

The `Str::substrReplace` method replaces text within a portion of a string, starting at the position specified by the third argument and replacing the number of characters specified by the fourth argument. Passing `0` to the method's fourth argument will insert the string at the specified position without replacing any of the existing characters in the string:

```
use Illuminate\Support\Str;

$result = Str::substrReplace('1300', ':', 2);
// 13:

$result = Str::substrReplace('1300', ':', 2, 0);
// 13:00
```

`Str::swap()`

The `Str::swap` method replaces multiple values in the given string using PHP's `strtr` function:

```
use Illuminate\Support\Str;

$string = Str::swap([
    'Tacos' => 'Burritos',
    'great' => 'fantastic',
], 'Tacos are great!');

// Burritos are fantastic!
```

Str::take()

The `Str::take` method returns a specified number of characters from the beginning of a string:

```
use Illuminate\Support\Str;

$taken = Str::take('Build something amazing!', 5);

// Build
```

Str::title()

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

```
use Illuminate\Support\Str;

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

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

Str::toBase64()

The `Str::toBase64` method converts the given string to Base64:

```
use Illuminate\Support\Str;

$base64 = Str::toBase64('Laravel');

// TGFyYXZlbA==
```

Str::toHtmlString()

The `Str::toHtmlString` method converts the string instance to an instance of `Illuminate\Support\HtmlString`, which may be displayed in Blade templates:

```
use Illuminate\Support\Str;

$htmlString = Str::of('Nuno Maduro')->toHtmlString();
```

Str::ucfirst()

The `Str::ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;

$string = Str::ucfirst('foo bar');

// Foo bar
```

Str::ucsplit()

The `Str::ucsplit` method splits the given string into an array by uppercase characters:

```
use Illuminate\Support\Str;

$segments = Str::ucsplit('FooBar');

// [0 => 'Foo', 1 => 'Bar']
```

Str::upper()

The `Str::upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$string = Str::upper('laravel');

// LARAVEL
```

Str::ulid()

The `Str::ulid` method generates a ULID, which is a compact, time-ordered unique identifier:

```
use Illuminate\Support\Str;

return (string) Str::ulid();

// 01gd6r360bp37zj17nxb55yv40
```

If you would like to retrieve a `Illuminate\Support\Carbon` date instance representing the date and time that a given ULID was created, you may use the `createFromId` method provided by Laravel's Carbon integration:

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

$date = Carbon::createFromId((string) Str::ulid());
```

During testing, it may be useful to "fake" the value that is returned by the `Str::ulid` method. To accomplish this, you may use the `createUlidsUsing` method:

```
use Symfony\Component\Uid\Ulid;

Str::createUlidsUsing(function () {
    return new Ulid('01HRDBNHHCKNW2AK4Z29SN82T9');
});
```

To instruct the `ulid` method to return to generating ULIDs normally, you may invoke the `createUlidsNormally` method:

```
Str::createUlidsNormally();
```

Str::unwrap()

The `Str::unwrap` method removes the specified strings from the beginning and end of a given string:

```
use Illuminate\Support\Str;

Str::unwrap('-Laravel-', '-');

// Laravel

Str::unwrap('{framework: "Laravel"}', '{', '}');

// framework: "Laravel"
```

Str::uuid()

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

```
use Illuminate\Support\Str;

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

During testing, it may be useful to "fake" the value that is returned by the `Str::uuid` method. To accomplish this, you may use the `createUuidsUsing` method:

```
use Ramsey\Uuid\Uuid;

Str::createUuidsUsing(function () {
    return Uuid::fromString('eadbfeac-5258-45c2-bab7-ccb9b5ef74f9');
});
```

To instruct the `uuid` method to return to generating UUIDs normally, you may invoke the `createUuidsNormally` method:

```
Str::createUuidsNormally();
```

Str::wordCount()

The `Str::wordCount` method returns the number of words that a string contains:

```
use Illuminate\Support\Str;

Str::wordCount('Hello, world!'); // 2
```

Str::wordWrap()

The `Str::wordWrap` method wraps a string to a given number of characters:

```
use Illuminate\Support\Str;

$text = "The quick brown fox jumped over the lazy dog.";

Str::wordWrap($text, characters: 20, break: "<br />\n");

/*
The quick brown fox<br />
jumped over the lazy<br />
dog.
*/
```

Str::words()

The `Str::words` method limits the number of words in a string. An additional string may be passed to this method via its third argument to specify which string should be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

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

// Perfectly balanced, as >>
```

Str::wrap()

The `Str::wrap` method wraps the given string with an additional string or pair of strings:

```
use Illuminate\Support\Str;

Str::wrap('Laravel', '');

// "Laravel"

Str::wrap('is', before: 'This ', after: ' Laravel!');

// This is Laravel!
```

str()

The `str` function returns a new `Illuminate\Support\Stringable` instance of the given string. This function is equivalent to the `Str::of` method:

```
$string = str('Taylor')->append(' Otwell');

// 'Taylor Otwell'
```

If no argument is provided to the `str` function, the function returns an instance of `Illuminate\Support\Str`:

```
$snake = str()->snake('FooBar');

// 'foo_bar'
```

trans()

The `trans` function translates the given translation key using your [language files](#):

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

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

`trans_choice()`

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

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

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

Fluent Strings

Fluent strings provide a more fluent, object-oriented interface for working with string values, allowing you to chain multiple string operations together using a more readable syntax compared to traditional string operations.

`after`

The `after` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->after('This is');

// ' my name'
```

`afterLast`

The `afterLast` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('App\Http\Controllers\Controller')->afterLast('\\');

// 'Controller'
```

`apa`

The `apa` method converts the given string to title case following the [APA guidelines](#):

```
use Illuminate\Support\Str;

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

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

`append`

The `append` method appends the given values to the string:

```
use Illuminate\Support\Str;

$string = Str::of('Taylor')->append(' Otwell');

// 'Taylor Otwell'
```

ascii

The `ascii` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$string = Str::of('ü')->ascii();

// 'u'
```

basename

The `basename` method will return the trailing name component of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->basename();

// 'baz'
```

If needed, you may provide an "extension" that will be removed from the trailing component:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz.jpg')->basename('.jpg');

// 'baz'
```

before

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

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->before('my name');

// 'This is '
```

beforeLast

The `beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->beforeLast('is');

// 'This '
```

between

The `between` method returns the portion of a string between two values:

```
use Illuminate\Support\Str;

$converted = Str::of('This is my name')->between('This', 'name');

// ' is my '
```

betweenFirst

The `betweenFirst` method returns the smallest possible portion of a string between two values:

```
use Illuminate\Support\Str;

$converted = Str::of('[a] bc [d]')->betweenFirst('[', ']');

// 'a'
```

camel

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

```
use Illuminate\Support\Str;

$converted = Str::of('foo_bar')->camel();

// 'fooBar'
```

charAt

The `charAt` method returns the character at the specified index. If the index is out of bounds, `false` is returned:

```
use Illuminate\Support\Str;

$character = Str::of('This is my name.')->charAt(6);

// 's'
```

classBasename

The `classBasename` method returns the class name of the given class with the class's namespace removed:

```
use Illuminate\Support\Str;

$string = Str::of('Foo\Bar\Baz')->class_basename();

// 'Baz'
```

contains

The `contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains('my');

// true
```

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

```
use Illuminate\Support\Str;

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

// true
```

containsAll

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

```
use Illuminate\Support\Str;

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

// true
```

dirname

The `dirname` method returns the parent directory portion of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname();

// '/foo/bar'
```

If necessary, you may specify how many directory levels you wish to trim from the string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname(2);

// '/foo'
```

excerpt

The `excerpt` method extracts an excerpt from the string that matches the first instance of a phrase within that string:

```
use Illuminate\Support\Str;

$excerpt = Str::of('This is my name')->excerpt('my', [
    'radius' => 3
]);

// '...is my na...'
```

The `radius` option, which defaults to `100`, allows you to define the number of characters that should appear on each side of the truncated string.

In addition, you may use the `omission` option to change the string that will be prepended and appended to the truncated string:

```
use Illuminate\Support\Str;

$excerpt = Str::of('This is my name')->excerpt('name', [
    'radius' => 3,
    'omission' => '(...)'
]);

// '(...) my name'
```

endsWith

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

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith('name');

// true
```

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

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith(['name', 'foo']);

// true

$result = Str::of('This is my name')->endsWith(['this', 'foo']);

// false
```

exactly

The `exactly` method determines if the given string is an exact match with another string:

```
use Illuminate\Support\Str;

$result = Str::of('Laravel')->exactly('Laravel');

// true
```

explode

The `explode` method splits the string by the given delimiter and returns a collection containing each section of the split string:

```
use Illuminate\Support\Str;

$collection = Str::of('foo bar baz')->explode(' ');

// collect(['foo', 'bar', 'baz'])
```

finish

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

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->finish('/');

// this/string/

$adjusted = Str::of('this/string/')->finish('/');

// this/string/
```

headline

The `headline` method will convert strings delimited by casing, hyphens, or underscores into a space delimited string with each word's first letter capitalized:

```
use Illuminate\Support\Str;

$headline = Str::of('taylor_otwell')->headline();

// Taylor Otwell

$headline = Str::of('EmailNotificationSent')->headline();

// Email Notification Sent
```

inlineMarkdown

The `inlineMarkdown` method converts GitHub flavored Markdown into inline HTML using [CommonMark](#). However, unlike the `markdown` method, it does not wrap all generated HTML in a block-level element:

```
use Illuminate\Support\Str;

$html = Str::of('**Laravel**')->inlineMarkdown();

// <strong>Laravel</strong>
```

Markdown Security

By default, Markdown supports raw HTML, which will expose Cross-Site Scripting (XSS) vulnerabilities when used with raw user input. As per the [CommonMark Security documentation](#), you may use the `html_input` option to either escape or strip raw HTML, and the `allow_unsafe_links` option to specify whether to allow unsafe links. If you need to allow some raw HTML, you should pass your compiled Markdown through an HTML Purifier:

```
use Illuminate\Support\Str;

Str::of('Inject: <script>alert("Hello XSS!");</script>')->inlineMarkdown([
    'html_input' => 'strip',
    'allow_unsafe_links' => false,
]);

// Inject: alert("Hello XSS!");
```

is

The `is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values

```
use Illuminate\Support\Str;

$matches = Str::of('foobar')->is('foo*');

// true

$matches = Str::of('foobar')->is('baz*');

// false
```

isAscii

The `isAscii` method determines if a given string is an ASCII string:

```
use Illuminate\Support\Str;

$result = Str::of('Taylor')->isAscii();

// true

$result = Str::of('ü')->isAscii();

// false
```

isEmpty

The `isEmpty` method determines if the given string is empty:

```
use Illuminate\Support\Str;

$result = Str::of(' ')>trim()->isEmpty();

// true

$result = Str::of('Laravel')>trim()->isEmpty();

// false
```

isNotEmpty

The `isNotEmpty` method determines if the given string is not empty:

```
use Illuminate\Support\Str;

$result = Str::of(' ')>trim()->isNotEmpty();

// false

$result = Str::of('Laravel')>trim()->isNotEmpty();

// true
```

isJson

The `isJson` method determines if a given string is valid JSON:

```
use Illuminate\Support\Str;

$result = Str::of('[1,2,3]')>isJson();

// true

$result = Str::of('{"first": "John", "last": "Doe"}')>isJson();

// true

$result = Str::of('{first: "John", last: "Doe"}')>isJson();

// false
```

isUlid

The `isUlid` method determines if a given string is a ULID:

```
use Illuminate\Support\Str;

$result = Str::of('01gd6r360bp37zj17nxb55yv40')->isUlid();

// true

$result = Str::of('Taylor')->isUlid();

// false
```

isUrl

The `isUrl` method determines if a given string is a URL:

```
use Illuminate\Support\Str;

$result = Str::of('http://example.com')->isUrl();

// true

$result = Str::of('Taylor')->isUrl();

// false
```

The `isUrl` method considers a wide range of protocols as valid. However, you may specify the protocols that should be considered valid by providing them to the `isUrl` method:

```
$result = Str::of('http://example.com')->isUrl(['http', 'https']);
```

isUuid

The `isUuid` method determines if a given string is a UUID:

```
use Illuminate\Support\Str;

$result = Str::of('5ace9ab9-e9cf-4ec6-a19d-5881212a452c')->isUuid();

// true

$result = Str::of('Taylor')->isUuid();

// false
```

kebab

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

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->kebab();

// foo-bar
```

lcfirst

The `lcfirst` method returns the given string with the first character lowercased:

```
use Illuminate\Support\Str;

$string = Str::of('Foo Bar')->lcfirst();

// foo Bar
```

length

The `length` method returns the length of the given string:

```
use Illuminate\Support\Str;

$length = Str::of('Laravel')->length();

// 7
```

limit

The `limit` method truncates the given string to the specified length:

```
use Illuminate\Support\Str;

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

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

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

```
use Illuminate\Support\Str;

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

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

lower

The `lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$result = Str::of('LARAVEL')->lower();

// 'laravel'
```

ltrim

The `ltrim` method trims the left side of the string:

```
use Illuminate\Support\Str;

$string = Str::of(' Laravel ')->ltrim();

// 'Laravel'

$string = Str::of('/Laravel/')->ltrim('/');

// 'Laravel/'
```

markdown

The `markdown` method converts GitHub flavored Markdown into HTML:

```
use Illuminate\Support\Str;

$html = Str::of('# Laravel')->markdown();

// <h1>Laravel</h1>

$html = Str::of('# Taylor <b>Otwell</b>')->markdown([
    'html_input' => 'strip',
]);

// <h1>Taylor Otwell</h1>
```

Markdown Security

By default, Markdown supports raw HTML, which will expose Cross-Site Scripting (XSS) vulnerabilities when used with raw user input. As per the [CommonMark Security documentation](#), you may use the `html_input` option to either escape or strip raw HTML, and the `allow_unsafe_links` option to specify whether to allow unsafe links. If you need to allow some raw HTML, you should pass your compiled Markdown through an HTML Purifier:

```
use Illuminate\Support\Str;

Str::of('Inject: <script>alert("Hello XSS!");</script>')->markdown([
    'html_input' => 'strip',
    'allow_unsafe_links' => false,
]);

// <p>Inject: alert("Hello XSS!");</p>
```

mask

The `mask` method masks a portion of a string with a repeated character, and may be used to obfuscate segments of strings such as email addresses and phone numbers:

```
use Illuminate\Support\Str;

$string = Str::of('taylor@example.com')->mask('*', 3);

// tay*****
```

If needed, you may provide negative numbers as the third or fourth argument to the `mask` method, which will instruct the method to begin masking at the given distance from the end of the string:

```
$string = Str::of('taylor@example.com')->mask('*', -15, 3);

// tay***@example.com

$string = Str::of('taylor@example.com')->mask('*', 4, -4);

// tay*****.com
```

match

The `match` method will return the portion of a string that matches a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('foo bar')->match('/bar/');

// 'bar'

$result = Str::of('foo bar')->match('/foo (.*?)/');

// 'bar'
```

matchAll

The `matchAll` method will return a collection containing the portions of a string that match a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('bar foo bar')->matchAll('/bar/');

// collect(['bar', 'bar'])
```

If you specify a matching group within the expression, Laravel will return a collection of that group's matches:

```
use Illuminate\Support\Str;

$result = Str::of('bar fun bar fly')->matchAll('/f(\w*)/');

// collect(['un', 'ly']);
```

If no matches are found, an empty collection will be returned.

isMatch

The `isMatch` method will return `true` if the string matches a given regular expression:

```
use Illuminate\Support\Str;

$result = Str::of('foo bar')->isMatch('/foo (.*)/');
// true

$result = Str::of('laravel')->isMatch('/foo (.*)/');
// false
```

newLine

The `newLine` method appends an "end of line" character to a string:

```
use Illuminate\Support\Str;

$padded = Str::of('Laravel')->newLine()->append('Framework');

// 'Laravel
// Framework'
```

padBoth

The `padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padBoth(10, '_');

// '__James__'

$padded = Str::of('James')->padBoth(10);

// ' James '
```

padLeft

The `padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padLeft(10, '-=');

// '====James'

$padded = Str::of('James')->padLeft(10);

// '      James'
```

padRight

The `padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padRight(10, '-');

// 'James-----'

$padded = Str::of('James')->padRight(10);

// 'James      '
```

pipe

The `pipe` method allows you to transform the string by passing its current value to the given callable:

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

$hash = Str::of('Laravel')->pipe('md5')->prepend('Checksum: ');

// 'Checksum: a5c95b86291ea299fcbe64458ed12702'

$closure = Str::of('foo')->pipe(function (Stringable $str) {
    return 'bar';
});

// 'bar'
```

plural

The `plural` method converts a singular word string to its plural form. This function supports [any of the languages support by Laravel's pluralizer](#):

```
use Illuminate\Support\Str;

$plural = Str::of('car')->plural();

// cars

$plural = Str::of('child')->plural();

// children
```

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

```
use Illuminate\Support\Str;

$plural = Str::of('child')->plural(2);

// children

$plural = Str::of('child')->plural(1);

// child
```

position

The `position` method returns the position of the first occurrence of a substring in a string. If the substring does not exist within the string, `false` is returned:

```
use Illuminate\Support\Str;

$position = Str::of('Hello, World!')->position('Hello');

// 0

$position = Str::of('Hello, World!')->position('W');

// 7
```

prepend

The `prepend` method prepends the given values onto the string:

```
use Illuminate\Support\Str;

$string = Str::of('Framework')->prepend('Laravel ');

// Laravel Framework
```

remove

The `remove` method removes the given value or array of values from the string:

```
use Illuminate\Support\Str;

$string = Str::of('Arkansas is quite beautiful!')->remove('quite');

// Arkansas is beautiful!
```

You may also pass `false` as a second parameter to ignore case when removing strings.

repeat

The `repeat` method repeats the given string:

```
use Illuminate\Support\Str;

$repeated = Str::of('a')->repeat(5);

// aaaaa
```

replace

The `replace` method replaces a given string within the string:

```
use Illuminate\Support\Str;

$replaced = Str::of('Laravel 6.x')->replace('6.x', '7.x');

// Laravel 7.x
```

The `replace` method also accepts a `caseSensitive` argument. By default, the `replace` method is case sensitive:

```
$replaced = Str::of('macOS 13.x')->replace(
    'macOS', 'iOS', caseSensitive: false
);
```

replaceArray

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

```
use Illuminate\Support\Str;

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

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

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

replaceFirst

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

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceFirst('the', 'a');

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

replaceLast

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

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceLast('the', 'a');

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

replaceMatches

The `replaceMatches` method replaces all portions of a string matching a pattern with the given replacement string:

```
use Illuminate\Support\Str;

$replaced = Str::of('(+1) 501-555-1000')->replaceMatches('/[^A-Za-z0-9]+/', '');

// '15015551000'
```

The `replaceMatches` method also accepts a closure that will be invoked with each portion of the string matching the given pattern, allowing you to perform the replacement logic within the closure and return the replaced value:

```
use Illuminate\Support\Str;

$replaced = Str::of('123')->replaceMatches('/\d/', function (array $matches) {
    return '['.$matches[0].']';
});

// '[1][2][3]'
```

replaceStart

The `replaceStart` method replaces the first occurrence of the given value only if the value appears at the start of the string:

```
use Illuminate\Support\Str;

$replaced = Str::of('Hello World')->replaceStart('Hello', 'Laravel');

// Laravel World

$replaced = Str::of('Hello World')->replaceStart('World', 'Laravel');

// Hello Laravel
```

replaceEnd

The `replaceEnd` method replaces the last occurrence of the given value only if the value appears at the end of the string:

```
use Illuminate\Support\Str;

$replaced = Str::of('Hello World')->replaceEnd('World', 'Laravel');

// Hello Laravel

$replaced = Str::of('Hello World')->replaceEnd('Hello', 'Laravel');

// Hello World
```

rtrim

The `rtrim` method trims the right side of the given string:

```
use Illuminate\Support\Str;

$string = Str::of(' Laravel ')->rtrim();

// ' Laravel'

$string = Str::of('/Laravel/')->rtrim('/');

// '/Laravel'
```

scan

The `scan` method parses input from a string into a collection according to a format supported by the [sscanf](#) PHP function:

```
use Illuminate\Support\Str;

$collection = Str::of('filename.jpg')->scan('%[^.].%s');

// collect(['filename', 'jpg'])
```

singular

The `singular` method converts a string to its singular form. This function supports [any of the languages support by Laravel's pluralizer](#):

```
use Illuminate\Support\Str;

$singular = Str::of('cars')->singular();

// car

$singular = Str::of('children')->singular();

// child
```

slug

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

```
use Illuminate\Support\Str;

$slug = Str::of('Laravel Framework')->slug('-');

// laravel-framework
```

snake

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

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->snake();

// foo_bar
```

split

The `split` method splits a string into a collection using a regular expression:

```
use Illuminate\Support\Str;

$segments = Str::of('one, two, three')->split('/[\s,]+/');
// collect(["one", "two", "three"])
```

squish

The `squish` method removes all extraneous white space from a string, including extraneous white space between words:

```
use Illuminate\Support\Str;

$string = Str::of('    laravel    framework    ')->squish();
// laravel framework
```

start

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

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->start('/');
// /this/string

$adjusted = Str::of('/this/string')->start('/');
// /this/string
```

startsWith

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

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->startsWith('This');
// true
```

stripTags

The `stripTags` method removes all HTML and PHP tags from a string:

```
use Illuminate\Support\Str;

$result = Str::of('<a href="https://laravel.com">Taylor <b>Otwell</b></a>')->stripTags();
// Taylor Otwell

$result = Str::of('<a href="https://laravel.com">Taylor <b>Otwell</b></a>')->stripTags('<b>');
// Taylor <b>Otwell</b>
```

studly

The `studly` method converts the given string to `StudyCase`:

```
use Illuminate\Support\Str;

$converted = Str::of('foo_bar')->studly();

// FooBar
```

substr

The `substr` method returns the portion of the string specified by the given start and length parameters:

```
use Illuminate\Support\Str;

$string = Str::of('Laravel Framework')->substr(8);

// Framework

$string = Str::of('Laravel Framework')->substr(8, 5);

// Frame
```

substrReplace

The `substrReplace` method replaces text within a portion of a string, starting at the position specified by the second argument and replacing the number of characters specified by the third argument. Passing `0` to the method's third argument will insert the string at the specified position without replacing any of the existing characters in the string:

```
use Illuminate\Support\Str;

$string = Str::of('1300')->substrReplace(':', 2);

// 13:

$string = Str::of('The Framework')->substrReplace(' Laravel', 3, 0);

// The Laravel Framework
```

swap

The `swap` method replaces multiple values in the string using PHP's `strtr` function:

```
use Illuminate\Support\Str;

$string = Str::of('Tacos are great!')
->swap([
    'Tacos' => 'Burritos',
    'great' => 'fantastic',
]);

// Burritos are fantastic!
```

take

The `take` method returns a specified number of characters from the beginning of the string:

```
use Illuminate\Support\Str;

$taken = Str::of('Build something amazing!')->take(5);

// Build
```

tap

The `tap` method passes the string to the given closure, allowing you to examine and interact with the string while not affecting the string itself. The original string is returned by the `tap` method regardless of what is returned by the closure:

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

$string = Str::of('Laravel')
    ->append(' Framework')
    ->tap(function (Stringable $string) {
        dump('String after append: '.$string);
    })
    ->upper();

// LARAVEL FRAMEWORK
```

test

The `test` method determines if a string matches the given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('Laravel Framework')->test('/Laravel/');

// true
```

title

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

```
use Illuminate\Support\Str;

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

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

toBase64()

The `toBase64` method converts the given string to Base64:

```
use Illuminate\Support\Str;

$base64 = Str::of('Laravel')->toBase64();

// TGFyYXZlbA==
```

trim

The `trim` method trims the given string:

```
use Illuminate\Support\Str;

$string = Str::of(' Laravel ')->trim();
// 'Laravel'

$string = Str::of('/Laravel/')->trim('/');
// 'Laravel'
```

ucfirst

The `ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;

$string = Str::of('foo bar')->ucfirst();
// Foo bar
```

ucssplit

The `ucssplit` method splits the given string into a collection by uppercase characters:

```
use Illuminate\Support\Str;

$string = Str::of('Foo Bar')->ucssplit();
// collect(['Foo', 'Bar'])
```

unwrap

The `unwrap` method removes the specified strings from the beginning and end of a given string:

```
use Illuminate\Support\Str;

Str::of('-Laravel-')->unwrap('-');
// Laravel

Str::of('{framework: "Laravel"}')->unwrap('{', '}');
// framework: "Laravel"
```

upper

The `upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$adjusted = Str::of('laravel')->upper();

// LARAVEL
```

when

The `when` method invokes the given closure if a given condition is `true`. The closure will receive the fluent string instance:

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

$string = Str::of('Taylor')
    ->when(true, function (Stringable $string) {
        return $string->append(' Otwell');
    });

// 'Taylor Otwell'
```

If necessary, you may pass another closure as the third parameter to the `when` method. This closure will execute if the condition parameter evaluates to `false`.

whenContains

The `whenContains` method invokes the given closure if the string contains the given value. The closure will receive the fluent string instance:

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

$string = Str::of('tony stark')
    ->whenContains('tony', function (Stringable $string) {
        return $string->title();
    });

// 'Tony Stark'
```

If necessary, you may pass another closure as the third parameter to the `when` method. This closure will execute if the string does not contain the given value.

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

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

$string = Str::of('tony stark')
    ->whenContains(['tony', 'hulk'], function (Stringable $string) {
        return $string->title();
    });

// Tony Stark
```

whenContainsAll

The `whenContainsAll` method invokes the given closure if the string contains all of the given sub-strings. The closure will receive the fluent string instance:

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

$string = Str::of('tony stark')
    ->whenContainsAll(['tony', 'stark'], function (Stringable $string) {
        return $string->title();
});

// 'Tony Stark'
```

If necessary, you may pass another closure as the third parameter to the `when` method. This closure will execute if the condition parameter evaluates to `false`.

whenEmpty

The `whenEmpty` method invokes the given closure if the string is empty. If the closure returns a value, that value will also be returned by the `whenEmpty` method. If the closure does not return a value, the fluent string instance will be returned:

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

$string = Str::of(' ')->whenEmpty(function (Stringable $string) {
    return $string->trim()->prepend('Laravel');
});

// 'Laravel'
```

whenNotEmpty

The `whenNotEmpty` method invokes the given closure if the string is not empty. If the closure returns a value, that value will also be returned by the `whenNotEmpty` method. If the closure does not return a value, the fluent string instance will be returned:

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

$string = Str::of('Framework')->whenNotEmpty(function (Stringable $string) {
    return $string->prepend('Laravel ');
});

// 'Laravel Framework'
```

whenStartsWith

The `whenStartsWith` method invokes the given closure if the string starts with the given sub-string. The closure will receive the fluent string instance:

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

$string = Str::of('disney world')->whenStartsWith('disney', function (Stringable $string) {
    return $string->title();
});

// 'Disney World'
```

whenEndsWith

The `whenEndsWith` method invokes the given closure if the string ends with the given sub-string. The closure will receive the fluent string instance:

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

$string = Str::of('disney world')->whenEndsWith('world', function (Stringable $string) {
    return $string->title();
});

// 'Disney World'
```

whenExactly

The `whenExactly` method invokes the given closure if the string exactly matches the given string. The closure will receive the fluent string instance:

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

$string = Str::of('laravel')->whenExactly('laravel', function (Stringable $string) {
    return $string->title();
});

// 'Laravel'
```

whenNotExactly

The `whenNotExactly` method invokes the given closure if the string does not exactly match the given string. The closure will receive the fluent string instance:

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

$string = Str::of('framework')->whenNotExactly('laravel', function (Stringable $string) {
    return $string->title();
});

// 'Framework'
```

whenIs

The `whenIs` method invokes the given closure if the string matches a given pattern. Asterisks may be used as wildcard values. The closure will receive the fluent string instance:

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

$string = Str::of('foo/bar')->whenIs('foo/*', function (Stringable $string) {
    return $string->append('/baz');
});

// 'foo/bar/baz'
```

whenIsAscii

The `whenIsAscii` method invokes the given closure if the string is 7 bit ASCII. The closure will receive the fluent string instance:

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

$string = Str::of('laravel')->whenIsAscii(function (Stringable $string) {
    return $string->title();
});

// 'Laravel'
```

whenIsUlid

The `whenIsUlid` method invokes the given closure if the string is a valid ULID. The closure will receive the fluent string instance:

```
use Illuminate\Support\Str;

$string = Str::of('01gd6r360bp37zj17nxb55yv40')->whenIsUlid(function (Stringable $string) {
    return $string->substr(0, 8);
});

// '01gd6r36'
```

whenIsUuid

The `whenIsUuid` method invokes the given closure if the string is a valid UUID. The closure will receive the fluent string instance:

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

$string = Str::of('a0a2a2d2-0b87-4a18-83f2-2529882be2de')->whenIsUuid(function (Stringable $string) {
    return $string->substr(0, 8);
});

// 'a0a2a2d2'
```

whenTest

The `whenTest` method invokes the given closure if the string matches the given regular expression. The closure will receive the fluent string instance:

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

$string = Str::of('laravel framework')->whenTest('/laravel/', function (Stringable $string) {
    return $string->title();
});

// 'Laravel Framework'
```

wordCount

The `wordCount` method returns the number of words that a string contains:

```
use Illuminate\Support\Str;

Str::of('Hello, world!')->wordCount(); // 2
```

words

The `words` method limits the number of words in a string. If necessary, you may specify an additional string that will be appended to the truncated string:

```
use Illuminate\Support\Str;

$string = Str::of('Perfectly balanced, as all things should be.')->words(3, ' >>>');

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

Task Scheduling

- [Introduction](#)
- [Defining Schedules](#)
 - [Scheduling Artisan Commands](#)
 - [Scheduling Queued Jobs](#)
 - [Scheduling Shell Commands](#)
 - [Schedule Frequency Options](#)
 - [Timezones](#)
 - [Preventing Task Overlaps](#)
 - [Running Tasks on One Server](#)
 - [Background Tasks](#)
 - [Maintenance Mode](#)
- [Running the Scheduler](#)
 - [Sub-Minute Scheduled Tasks](#)
 - [Running the Scheduler Locally](#)
- [Task Output](#)
- [Task Hooks](#)
- [Events](#)

Introduction

In the past, you may have written a cron configuration entry for each task you needed to schedule on your server. However, this can quickly become a pain because your task schedule is no longer in source control and you must SSH into your server to view your existing cron entries or add additional entries.

Laravel's command scheduler offers a fresh approach to managing scheduled tasks on your server. The scheduler allows you to fluently and expressively define your command schedule within your Laravel application itself. When using the scheduler, only a single cron entry is needed on your server. Your task schedule is defined in the `app\Console\Kernel.php` file's `schedule` method. To help you get started, a simple example is defined within the method.

Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of your application's `App\Console\Kernel` class. To get started, let's take a look at an example. In this example, we will schedule a closure to be called every day at midnight. Within the closure we will execute a database query to clear a table:

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     */
    protected function schedule(Schedule $schedule): void
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

In addition to scheduling using closures, you may also schedule [invokable objects](#). Invokable objects are simple PHP classes that contain an `__invoke` method:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

If you would like to view an overview of your scheduled tasks and the next time they are scheduled to run, you may use the `schedule:list` Artisan command:

```
php artisan schedule:list
```

Scheduling Artisan Commands

In addition to scheduling closures, you may also schedule [Artisan commands](#) and system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class.

When scheduling Artisan commands using the command's class name, you may pass an array of additional command-line arguments that should be provided to the command when it is invoked:

```
use App\Console\Commands\SendEmailsCommand;

$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(SendEmailsCommand::class, ['Taylor', '--force'])->daily();
```

Scheduling Queued Jobs

The `job` method may be used to schedule a [queued job](#). This method provides a convenient way to schedule queued jobs without using the `call` method to define closures to queue the job:

```
use App\Jobs\Heartbeat;

$schedule->job(new Heartbeat)->everyFiveMinutes();
```

Optional second and third arguments may be provided to the `job` method which specifies the queue name and queue connection that should be used to queue the job:

```
use App\Jobs\Heartbeat;

// Dispatch the job to the "heartbeats" queue on the "sq" connection...
$schedule->job(new Heartbeat, 'heartbeats', 'sq')->everyFiveMinutes();
```

Scheduling Shell Commands

The `exec` method may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

We've already seen a few examples of how you may configure a task to run at specified intervals. However, there are many more task schedule frequencies that you may assign to a task:

Method	Description
<code>->cron('* * * * *');</code>	Run the task on a custom cron schedule
<code>->everySecond();</code>	Run the task every second
<code>->everyTwoSeconds();</code>	Run the task every two seconds
<code>->everyFiveSeconds();</code>	Run the task every five seconds
<code>->everyTenSeconds();</code>	Run the task every ten seconds
<code>->everyFifteenSeconds();</code>	Run the task every fifteen seconds
<code>->everyTwentySeconds();</code>	Run the task every twenty seconds
<code>->everyThirtySeconds();</code>	Run the task every thirty seconds
<code>->everyMinute();</code>	Run the task every minute
<code>->everyTwoMinutes();</code>	Run the task every two minutes
<code>->everyThreeMinutes();</code>	Run the task every three minutes
<code>->everyFourMinutes();</code>	Run the task every four minutes
<code>->everyFiveMinutes();</code>	Run the task every five minutes
<code>->everyTenMinutes();</code>	Run the task every ten minutes
<code>->everyFifteenMinutes();</code>	Run the task every fifteen minutes
<code>->everyThirtyMinutes();</code>	Run the task every thirty minutes
<code>->hourly();</code>	Run the task every hour
<code>->hourlyAt(17);</code>	Run the task every hour at 17 minutes past the hour
<code>->everyOddHour(\$minutes = 0);</code>	Run the task every odd hour
<code>->everyTwoHours(\$minutes = 0);</code>	Run the task every two hours
<code>->everyThreeHours(\$minutes = 0);</code>	Run the task every three hours
<code>->everyFourHours(\$minutes = 0);</code>	Run the task every four hours
<code>->everySixHours(\$minutes = 0);</code>	Run the task every six hours
<code>->daily();</code>	Run the task every day at midnight
<code>->dailyAt('13:00');</code>	Run the task every day at 13:00
<code>->twiceDaily(1, 13);</code>	Run the task daily at 1:00 & 13:00
<code>->twiceDailyAt(1, 13, 15);</code>	Run the task daily at 1:15 & 13:15
<code>->weekly();</code>	Run the task every Sunday at 00:00
<code>->weeklyOn(1, '8:00');</code>	Run the task every week on Monday at 8:00
<code>->monthly();</code>	Run the task on the first day of every month at 00:00
<code>->monthlyOn(4, '15:00');</code>	Run the task every month on the 4th at 15:00

<code>->twiceMonthly(1, 16, '13:00');</code>	Run the task monthly on the 1st and 16th at 13:00
<code>->lastDayOfMonth('15:00');</code>	Run the task on the last day of the month at 15:00
<code>->quarterly();</code>	Run the task on the first day of every quarter at 00:00
<code>->quarterlyOn(4, '14:00');</code>	Run the task every quarter on the 4th at 14:00
<code>->yearly();</code>	Run the task on the first day of every year at 00:00
<code>->yearlyOn(6, 1, '17:00');</code>	Run the task every year on June 1st at 17:00
<code>->timezone('America/New_York');</code>	Set the timezone for the task

These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, you may schedule a command to run weekly on Monday:

```
// Run once per week on Monday at 1 PM...
$schedule->call(function () {
    // ...
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')
    ->weekdays()
    ->hourly()
    ->timezone('America/Chicago')
    ->between('8:00', '17:00');
```

A list of additional schedule constraints may be found below:

Method	Description
<code>->weekdays();</code>	Limit the task to weekdays
<code>->weekends();</code>	Limit the task to weekends
<code>->sundays();</code>	Limit the task to Sunday
<code>->mondays();</code>	Limit the task to Monday
<code>->tuesdays();</code>	Limit the task to Tuesday
<code>->wednesdays();</code>	Limit the task to Wednesday
<code>->thursdays();</code>	Limit the task to Thursday
<code>->fridays();</code>	Limit the task to Friday
<code>->saturdays();</code>	Limit the task to Saturday
<code>->days(array mixed);</code>	Limit the task to specific days
<code>->between(\$startTime, \$endTime);</code>	Limit the task to run between start and end times
<code>->unlessBetween(\$startTime, \$endTime);</code>	Limit the task to not run between start and end times
<code>->when(Closure);</code>	Limit the task based on a truth test
<code>->environments(\$env);</code>	Limit the task to specific environments

Day Constraints

The `days` method may be used to limit the execution of a task to specific days of the week. For example, you may schedule a command to run hourly on Sundays and Wednesdays:

```
$schedule->command('emails:send')
    ->hourly()
    ->days([0, 3]);
```

Alternatively, you may use the constants available on the `Illuminate\Console\Scheduling\Schedule` class when defining the days on which a task should run:

```
use Illuminate\Console\Scheduling\Schedule;

$schedule->command('emails:send')
    ->hourly()
    ->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

Between Time Constraints

The `between` method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('emails:send')
    ->hourly()
    ->between('7:00', '22:00');
```

Similarly, the `unlessBetween` method can be used to exclude the execution of a task for a period of time:

```
$schedule->command('emails:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

Truth Test Constraints

The `when` method may be used to limit the execution of a task based on the result of a given truth test. In other words, if the given closure returns `true`, the task will execute as long as no other constraining conditions prevent the task from running:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

The `skip` method may be seen as the inverse of `when`. If the `skip` method returns `true`, the scheduled task will not be executed:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return `true`.

Environment Constraints

The `environments` method may be used to execute tasks only on the given environments (as defined by the `APP_ENV` environment variable):

```
$schedule->command('emails:send')
    ->daily()
    ->environments(['staging', 'production']);
```

Timezones

Using the `timezone` method, you may specify that a scheduled task's time should be interpreted within a given timezone:

```
$schedule->command('report:generate')
    ->timezone('America/New_York')
    ->at('2:00')
```

If you are repeatedly assigning the same timezone to all of your scheduled tasks, you may wish to define a `scheduleTimezone` method in your `App\Console\Kernel` class. This method should return the default timezone that should be assigned to all scheduled tasks:

```
use DateTimeZone;

/**
 * Get the timezone that should be used by default for scheduled events.
 */
protected function scheduleTimezone(): DateTimeZone|string|null
{
    return 'America/Chicago';
}
```



Remember that some timezones utilize daylight savings time. When daylight saving time changes occur, your scheduled task may run twice or even not run at all. For this reason, we recommend avoiding timezone scheduling when possible.

Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` Artisan command will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

If needed, you may specify how many minutes must pass before the "without overlapping" lock expires. By default, the lock will expire after 24 hours:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

Behind the scenes, the `withoutOverlapping` method utilizes your application's `cache` to obtain locks. If necessary, you can clear these cache locks using the `schedule:clear-cache` Artisan command. This is typically only necessary if a task becomes stuck due to an unexpected server problem.

Running Tasks on One Server



To utilize this feature, your application must be using the `database`, `memcached`, `dynamodb`, or `redis` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

If your application's scheduler is running on multiple servers, you may limit a scheduled job to only execute on a single server. For instance, assume you have a scheduled task that generates a new report every Friday night. If the task scheduler is running on three worker servers, the scheduled task will run on all three servers and generate the report three times. Not good!

To indicate that the task should run on only one server, use the `onOneServer` method when defining the scheduled task. The first server to obtain the task will secure an atomic lock on the job to prevent other servers from running the same task at the same time:

```
$schedule->command('report:generate')
    ->fridays()
    ->at('17:00')
    ->onOneServer();
```

Naming Single Server Jobs

Sometimes you may need to schedule the same job to be dispatched with different parameters, while still instructing Laravel to run each permutation of the job on a single server. To accomplish this, you may assign each schedule definition a unique name via the `name` method:

```
$schedule->job(new CheckUptime('https://laravel.com'))
    ->name('check_uptime:laravel.com')
    ->everyFiveMinutes()
    ->onOneServer();

$schedule->job(new CheckUptime('https://vapor.laravel.com'))
    ->name('check_uptime:vapor.laravel.com')
    ->everyFiveMinutes()
    ->onOneServer();
```

Similarly, scheduled closures must be assigned a name if they are intended to be run on one server:

```
$schedule->call(fn () => User::resetApiRequestCount())
    ->name('reset-api-request-count')
    ->daily()
    ->onOneServer();
```

Background Tasks

By default, multiple tasks scheduled at the same time will execute sequentially based on the order they are defined in your `schedule` method. If you have long-running tasks, this may cause subsequent tasks to start much later than anticipated.

If you would like to run tasks in the background so that they may all run simultaneously, you may use the `runInBackground` method:

```
$schedule->command('analytics:report')
    ->daily()
    ->runInBackground();
```



The `runInBackground` method may only be used when scheduling tasks via the `command` and `exec` methods.

Maintenance Mode

Your application's scheduled tasks will not run when the application is in [maintenance mode](#), since we don't want your tasks to interfere with any unfinished maintenance you may be performing on your server. However, if you would like to force a task to run even in maintenance mode, you may call the `evenInMaintenanceMode` method when defining the task:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

Running the Scheduler

Now that we have learned how to define scheduled tasks, let's discuss how to actually run them on our server. The `schedule:run` Artisan command will evaluate all of your scheduled tasks and determine if they need to run based on the server's current time.

So, when using Laravel's scheduler, we only need to add a single cron configuration entry to our server that runs the `schedule:run` command every minute. If you do not know how to add cron entries to your server, consider using a service such as [Laravel Forge](#) which can manage the cron entries for you:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

Sub-Minute Scheduled Tasks

On most operating systems, cron jobs are limited to running a maximum of once per minute. However, Laravel's scheduler allows you to schedule tasks to run at more frequent intervals, even as often as once per second:

```
$schedule->call(function () {
    DB::table('recent_users')->delete();
})->everySecond();
```

When sub-minute tasks are defined within your application, the `schedule:run` command will continue running until the end of the current minute instead of exiting immediately. This allows the command to invoke all required sub-minute tasks throughout the minute.

Since sub-minute tasks that take longer than expected to run could delay the execution of later sub-minute tasks, it is recommend that all sub-minute tasks dispatch queued jobs or background commands to handle the actual task processing:

```
use App\Jobs\DeleteRecentUsers;

$schedule->job(new DeleteRecentUsers)->everyTenSeconds();

$schedule->command('users:delete')->everyTenSeconds()->runInBackground();
```

Interrupting Sub-Minute Tasks

As the `schedule:run` command runs for the entire minute of invocation when sub-minute tasks are defined, you may sometimes need to interrupt the command when deploying your application. Otherwise, an instance of the `schedule:run` command that is already running would continue using your application's previously deployed code until the current minute ends.

To interrupt in-progress `schedule:run` invocations, you may add the `schedule:interrupt` command to your application's deployment script. This command should be invoked after your application is finished deploying:

```
php artisan schedule:interrupt
```

Running the Scheduler Locally

Typically, you would not add a scheduler cron entry to your local development machine. Instead, you may use the `schedule:work` Artisan command. This command will run in the foreground and invoke the scheduler every minute until you terminate the command:

```
php artisan schedule:work
```

Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may email the output to an email address of your choice. Before emailing the output of a task, you should configure Laravel's [email services](#):

```
$schedule->command('report:generate')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('taylor@example.com');
```

If you only want to email the output if the scheduled Artisan or system command terminates with a non-zero exit code, use the `emailOutputOnFailure` method:

```
$schedule->command('report:generate')
    ->daily()
    ->emailOutputOnFailure('taylor@example.com');
```



The `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo`, and `appendOutputTo` methods are exclusive to the `command` and `exec` methods.

Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is executed:

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // The task is about to execute...
    })
    ->after(function () {
        // The task has executed...
});
```

The `onSuccess` and `onFailure` methods allow you to specify code to be executed if the scheduled task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
    ->daily()
    ->onSuccess(function () {
        // The task succeeded...
    })
    ->onFailure(function () {
        // The task failed...
});
```

If output is available from your command, you may access it in your `after`, `onSuccess` or `onFailure` hooks by type-hinting an `Illuminate\Support\Stringable` instance as the `$output` argument of your hook's closure definition:

```
use Illuminate\Support\Stringable;

$schedule->command('emails:send')
    ->daily()
    ->onSuccess(function (Stringable $output) {
        // The task succeeded...
    })
    ->onFailure(function (Stringable $output) {
        // The task failed...
});
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is executed. This method is useful for notifying an external service, such as [Envoyer](#), that your scheduled task is beginning or has finished execution:

```
$schedule->command('emails:send')
    ->daily()
    ->pingBefore($url)
    ->thenPing($url);
```

The `pingBeforeIf` and `thenPingIf` methods may be used to ping a given URL only if a given condition is `true`:

```
$schedule->command('emails:send')
    ->daily()
    ->pingBeforeIf($condition, $url)
    ->thenPingIf($condition, $url);
```

The `pingOnSuccess` and `pingOnFailure` methods may be used to ping a given URL only if the task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
    ->daily()
    ->pingOnSuccess($successUrl)
    ->pingOnFailure($failureUrl);
```

All of the ping methods require the Guzzle HTTP library. Guzzle is typically installed in all new Laravel projects by default, but, you may manually install Guzzle into your project using the Composer package manager if it has been accidentally removed:

```
composer require guzzlehttp/guzzle
```

Events

If needed, you may listen to `events` dispatched by the scheduler. Typically, event listener mappings will be defined within your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Console\Events\ScheduledTaskStarting' => [
        'App\Listeners\LogScheduledTaskStarting',
    ],
    'Illuminate\Console\Events\ScheduledTaskFinished' => [
        'App\Listeners\LogScheduledTaskFinished',
    ],
    'Illuminate\Console\Events\ScheduledBackgroundTaskFinished' => [
        'App\Listeners\LogScheduledBackgroundTaskFinished',
    ],
    'Illuminate\Console\Events\ScheduledTaskSkipped' => [
        'App\Listeners\LogScheduledTaskSkipped',
    ],
    'Illuminate\Console\Events\ScheduledTaskFailed' => [
        'App\Listeners\LogScheduledTaskFailed',
    ],
];
```

Chapter 6

Security

Authentication

- [Introduction](#)
 - [Starter Kits](#)
 - [Database Considerations](#)
 - [Ecosystem Overview](#)
- [Authentication Quickstart](#)
 - [Install a Starter Kit](#)
 - [Retrieving the Authenticated User](#)
 - [Protecting Routes](#)
 - [Login Throttling](#)
- [Manually Authenticating Users](#)
 - [Remembering Users](#)
 - [Other Authentication Methods](#)
- [HTTP Basic Authentication](#)
 - [Stateless HTTP Basic Authentication](#)
- [Logging Out](#)
 - [Invalidating Sessions on Other Devices](#)
- [Password Confirmation](#)
 - [Configuration](#)
 - [Routing](#)
 - [Protecting Routes](#)
- [Adding Custom Guards](#)
 - [Closure Request Guards](#)
- [Adding Custom User Providers](#)
 - [The User Provider Contract](#)
 - [The Authenticatable Contract](#)
- [Social Authentication](#)
- [Events](#)

Introduction

Many web applications provide a way for their users to authenticate with the application and "login". Implementing this feature in web applications can be a complex and potentially risky endeavor. For this reason, Laravel strives to give you the tools you need to implement authentication quickly, securely, and easily.

At its core, Laravel's authentication facilities are made up of "guards" and "providers". Guards define how users are authenticated for each request. For example, Laravel ships with a `session` guard which maintains state using session storage and cookies.

Providers define how users are retrieved from your persistent storage. Laravel ships with support for retrieving users using [Eloquent](#) and the database query builder. However, you are free to define additional providers as needed for your application.

Your application's authentication configuration file is located at `config/auth.php`. This file contains several well-documented options for tweaking the behavior of Laravel's authentication services.



Guards and providers should not be confused with "roles" and "permissions". To learn more about authorizing user actions via permissions, please refer to the [authorization](#) documentation.

Starter Kits

Want to get started fast? Install a [Laravel application starter kit](#) in a fresh Laravel application. After migrating your database, navigate your browser to `/register` or any other URL that is assigned to your application. The starter kits will take care of scaffolding your entire authentication system!

Even if you choose not to use a starter kit in your final Laravel application, installing the [Laravel Breeze](#) starter kit can be a wonderful opportunity to learn how to implement all of Laravel's authentication functionality in an actual Laravel project. Since Laravel Breeze creates authentication controllers, routes, and views for you, you can examine the code within these files to learn how Laravel's authentication features may be implemented.

Database Considerations

By default, Laravel includes an `App\Models\User` [Eloquent model](#) in your `app/Models` directory. This model may be used with the default Eloquent authentication driver. If your application is not using Eloquent, you may use the [database](#) authentication provider which uses the Laravel query builder.

When building the database schema for the `App\Models\User` model, make sure the password column is at least 60 characters in length. Of course, the `users` table migration that is included in new Laravel applications already creates a column that exceeds this length.

Also, you should verify that your `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for users that select the "remember me" option when logging into your application. Again, the default `users` table migration that is included in new Laravel applications already contains this column.

Ecosystem Overview

Laravel offers several packages related to authentication. Before continuing, we'll review the general authentication ecosystem in Laravel and discuss each package's intended purpose.

First, consider how authentication works. When using a web browser, a user will provide their username and password via a login form. If these credentials are correct, the application will store information about the authenticated user in the user's [session](#). A cookie issued to the browser contains the session ID so that subsequent requests to the application can associate the user with the correct session. After the session cookie is received, the application will retrieve the session data based on the session ID, note that the authentication information has been stored in the session, and will consider the user as "authenticated".

When a remote service needs to authenticate to access an API, cookies are not typically used for authentication because there is no web browser. Instead, the remote service sends an API token to the API on each request. The application may validate the incoming token against a table of valid API tokens and "authenticate" the request as being performed by the user associated with that API token.

Laravel's Built-in Browser Authentication Services

Laravel includes built-in authentication and session services which are typically accessed via the `Auth` and `Session` facades. These features provide cookie-based authentication for requests that are initiated from web browsers. They provide methods that allow you to verify a user's credentials and authenticate the user. In addition, these services will automatically store the proper authentication data in the user's session and issue the user's session cookie. A discussion of how to use these services is contained within this documentation.

Application Starter Kits

As discussed in this documentation, you can interact with these authentication services manually to build your application's own authentication layer. However, to help you get started more quickly, we have released [free packages](#) that provide robust, modern scaffolding of the entire authentication layer. These packages are [Laravel Breeze](#), [Laravel Jetstream](#), and [Laravel Fortify](#).

Laravel Breeze is a simple, minimal implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is comprised of simple [Blade templates](#) styled with [Tailwind CSS](#). To get started, check out the documentation on Laravel's [application starter kits](#).

Laravel Fortify is a headless authentication backend for Laravel that implements many of the features found in this documentation, including cookie-based authentication as well as other features such as two-factor authentication and email verification. Fortify provides the authentication backend for Laravel Jetstream or may be used independently in combination with [Laravel Sanctum](#) to provide authentication for an SPA that needs to authenticate with Laravel.

[Laravel Jetstream](#) is a robust application starter kit that consumes and exposes Laravel Fortify's authentication services with a beautiful, modern UI powered by [Tailwind CSS](#), [Livewire](#), and / or [Inertia](#). Laravel Jetstream includes optional support for two-factor authentication, team support, browser session management, profile management, and built-in integration with [Laravel Sanctum](#) to offer API token authentication. Laravel's API authentication offerings are discussed below.

Laravel's API Authentication Services

Laravel provides two optional packages to assist you in managing API tokens and authenticating requests made with API tokens: [Passport](#) and [Sanctum](#). Please note that these libraries and Laravel's built-in cookie based authentication libraries are not mutually exclusive. These libraries primarily focus on API token authentication while the built-in authentication services focus on cookie based browser authentication. Many applications will use both Laravel's built-in cookie based authentication services and one of Laravel's API authentication packages.

Passport

Passport is an OAuth2 authentication provider, offering a variety of OAuth2 "grant types" which allow you to issue various types of tokens. In general, this is a robust and complex package for API authentication. However, most applications do not require the complex features offered by the OAuth2 spec, which can be confusing for both users and developers. In addition, developers have been historically confused about how to authenticate SPA applications or mobile applications using OAuth2 authentication providers like Passport.

Sanctum

In response to the complexity of OAuth2 and developer confusion, we set out to build a simpler, more streamlined authentication package that could handle both first-party web requests from a web browser and API requests via tokens. This goal was realized with the release of [Laravel Sanctum](#), which should be considered the preferred and recommended authentication package for applications that will be offering a first-party web UI in addition to an API, or will be powered by a single-page application (SPA) that exists separately from the backend Laravel application, or applications that offer a mobile client.

Laravel Sanctum is a hybrid web / API authentication package that can manage your application's entire authentication process. This is possible because when Sanctum based applications receive a request, Sanctum will first determine if the request includes a session cookie that references an authenticated session. Sanctum accomplishes this by calling Laravel's built-in authentication services which we discussed earlier. If the request is not being authenticated via a session cookie, Sanctum will inspect the request for an API token. If an API token is present, Sanctum will authenticate the request using that token. To learn more about this process, please consult Sanctum's ["how it works"](#) documentation.

Laravel Sanctum is the API package we have chosen to include with the [Laravel Jetstream](#) application starter kit because we believe it is the best fit for the majority of web application's authentication needs.

Summary and Choosing Your Stack

In summary, if your application will be accessed using a browser and you are building a monolithic Laravel application, your application will use Laravel's built-in authentication services.

Next, if your application offers an API that will be consumed by third parties, you will choose between [Passport](#) or [Sanctum](#) to provide API token authentication for your application. In general, Sanctum should be preferred when possible since it is a simple, complete solution for API authentication, SPA authentication, and mobile authentication, including support for "scopes" or "abilities".

If you are building a single-page application (SPA) that will be powered by a Laravel backend, you should use [Laravel Sanctum](#). When using Sanctum, you will either need to [manually implement your own backend authentication routes](#) or utilize [Laravel Fortify](#) as a headless authentication backend service that provides routes and controllers for features such as registration, password reset, email verification, and more.

Passport may be chosen when your application absolutely needs all of the features provided by the OAuth2 specification.

And, if you would like to get started quickly, we are pleased to recommend [Laravel Breeze](#) as a quick way to start a new Laravel application that already uses our preferred authentication stack of Laravel's built-in authentication services and Laravel Sanctum.

Authentication Quickstart



This portion of the documentation discusses authenticating users via the [Laravel application starter kits](#), which includes UI scaffolding to help you get started quickly. If you would like to integrate with Laravel's authentication systems directly, check out the documentation on [manually authenticating users](#).

Install a Starter Kit

First, you should [install a Laravel application starter kit](#). Our current starter kits, Laravel Breeze and Laravel Jetstream, offer beautifully designed starting points for incorporating authentication into your fresh Laravel application.

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is made up of simple [Blade templates](#) styled with [Tailwind CSS](#). Additionally, Breeze provides scaffolding options based on [Livewire](#) or [Inertia](#), with the choice of using Vue or React for the Inertia-based scaffolding.

[Laravel Jetstream](#) is a more robust application starter kit that includes support for scaffolding your application with [Livewire](#) or [Inertia and Vue](#). In addition, Jetstream features optional support for two-factor authentication, teams, profile management, browser session management, API support via [Laravel Sanctum](#), account deletion, and more.

Retrieving the Authenticated User

After installing an authentication starter kit and allowing users to register and authenticate with your application, you will often need to interact with the currently authenticated user. While handling an incoming request, you may access the authenticated user via the `Auth` facade's `user` method:

```
use Illuminate\Support\Facades\Auth;

// Retrieve the currently authenticated user...
$user = Auth::user();

// Retrieve the currently authenticated user's ID...
$id = Auth::id();
```

Alternatively, once a user is authenticated, you may access the authenticated user via an `Illuminate\Http\Request` instance. Remember, type-hinted classes will automatically be injected into your controller methods. By type-hinting the `Illuminate\Http\Request` object, you may gain convenient access to the authenticated user from any controller method in your application via the request's `user` method:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Update the flight information for an existing flight.
     */
    public function update(Request $request): RedirectResponse
    {
        $user = $request->user();

        // ...

        return redirect('/flights');
    }
}
```

Determining if the Current User is Authenticated

To determine if the user making the incoming HTTP request is authenticated, you may use the `check` method on the `Auth` facade. This method will return `true` if the user is authenticated:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

 Even though it is possible to determine if a user is authenticated using the `check` method, you will typically use a middleware to verify that the user is authenticated before allowing the user access to certain routes / controllers. To learn more about this, check out the documentation on [protecting routes](#).

Protecting Routes

`Route middleware` can be used to only allow authenticated users to access a given route. Laravel ships with an `auth` middleware, which references the `Illuminate\Auth\Middleware\Authenticate` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth');
```

Redirecting Unauthenticated Users

When the `auth` middleware detects an unauthenticated user, it will redirect the user to the `login` [named route](#). You may modify this behavior by updating the `redirectTo` function in your application's `app/Http/Middleware/Authenticate.php` file:

```
use Illuminate\Http\Request;

/**
 * Get the path the user should be redirected to.
 */
protected function redirectTo(Request $request): string
{
    return route('login');
}
```

Specifying a Guard

When attaching the `auth` middleware to a route, you may also specify which "guard" should be used to authenticate the user. The guard specified should correspond to one of the keys in the `guards` array of your `auth.php` configuration file:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth:admin');
```

Login Throttling

If you are using the Laravel Breeze or Laravel Jetstream [starter kits](#), rate limiting will automatically be applied to login attempts. By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts. The throttling is unique to the user's username / email address and their IP address.



If you would like to rate limit other routes in your application, check out the [rate limiting documentation](#).

Manually Authenticating Users

You are not required to use the authentication scaffolding included with Laravel's [application starter kits](#). If you choose not to use this scaffolding, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the `Auth facade`, so we'll need to make sure to import the `Auth` facade at the top of the class. Next, let's check out the `attempt` method. The `attempt` method is normally used to handle authentication attempts from your application's "login" form. If authentication is successful, you should regenerate the user's [session](#) to prevent [session fixation](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     */
    public function authenticate(Request $request): RedirectResponse
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ])->onlyInput('email');
    }
}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the `password` value passed to the method via the array. You should not hash the incoming request's `password` value, since the framework will automatically hash the value before comparing it to the hashed password in the database. An authenticated session will be started for the user if the two hashed passwords match.

Remember, Laravel's authentication services will retrieve users from your database based on your authentication guard's "provider" configuration. In the default `config/auth.php` configuration file, the Eloquent user provider is specified and it is instructed to use the `App\Models\User` model when retrieving users. You may change these values within your configuration file based on the needs of your application.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

The `intended` method provided by Laravel's redirector will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware. A fallback URI may be given to this method in case the intended destination is not available.

Specifying Additional Conditions

If you wish, you may also add extra query conditions to the authentication query in addition to the user's email and password. To accomplish this, we may simply add the query conditions to the array passed to the `attempt` method. For example, we may verify that the user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // Authentication was successful...
}
```

For complex query conditions, you may provide a closure in your array of credentials. This closure will be invoked with the query instance, allowing you to customize the query based on your application's needs:

```
use Illuminate\Database\Eloquent\Builder;

if (Auth::attempt([
    'email' => $email,
    'password' => $password,
    fn (Builder $query) => $query->has('activeSubscription'),
])) {
    // Authentication was successful...
}
```



In these examples, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database table.

The `attemptWhen` method, which receives a closure as its second argument, may be used to perform more extensive inspection of the potential user before actually authenticating the user. The closure receives the potential user and should return `true` or `false` to indicate if the user may be authenticated:

```
if (Auth::attemptWhen([
    'email' => $email,
    'password' => $password,
], function (User $user) {
    return $user->isNotBanned();
})) {
    // Authentication was successful...
}
```

Accessing Specific Guard Instances

Via the `Auth` facade's `guard` method, you may specify which guard instance you would like to utilize when authenticating the user. This allows you to manage authentication for separate parts of your application using entirely separate authenticatable models or user tables.

The guard name passed to the `guard` method should correspond to one of the guards configured in your `auth.php` configuration file:

```
if (Auth::guard('admin')->attempt($credentials)) {
    // ...
}
```

Remembering Users

Many web applications provide a "remember me" checkbox on their login form. If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method.

When this value is `true`, Laravel will keep the user authenticated indefinitely or until they manually logout. Your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token. The `users` table migration included with new Laravel applications already includes this column:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

If your application offers "remember me" functionality, you may use the `viaRemember` method to determine if the currently authenticated user was authenticated using the "remember me" cookie:

```
use Illuminate\Support\Facades\Auth;

if (Auth::viaRemember()) {
    // ...
}
```

Other Authentication Methods

Authenticate a User Instance

If you need to set an existing user instance as the currently authenticated user, you may pass the user instance to the `Auth` facade's `login` method. The given user instance must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` contract. The `App\Models\User` model included with Laravel already implements this interface. This method of authentication is useful when you already have a valid user instance, such as directly after a user registers with your application:

```
use Illuminate\Support\Facades\Auth;

Auth::login($user);
```

You may pass a boolean value as the second argument to the `login` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::login($user, $remember = true);
```

If needed, you may specify an authentication guard before calling the `login` method:

```
Auth::guard('admin')->login($user);
```

Authenticate a User by ID

To authenticate a user using their database record's primary key, you may use the `loginUsingId` method. This method accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);
```

You may pass a boolean value as the second argument to the `loginUsingId` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::loginUsingId(1, $remember = true);
```

Authenticate a User Once

You may use the `once` method to authenticate a user with the application for a single request. No sessions or cookies will be utilized when calling this method:

```
if (Auth::once($credentials)) {
    // ...
}
```

HTTP Basic Authentication

[HTTP Basic Authentication](#) provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the `auth.basic` middleware to a route. The `auth.basic` middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('/profile', function () {
    // Only authenticated users may access this route...
})->middleware('auth.basic');
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the `auth.basic` middleware will assume the `email` column on your `users` database table is the user's "username".

A Note on FastCGI

If you are using PHP FastCGI and Apache to serve your Laravel application, HTTP Basic authentication may not work correctly. To correct these problems, the following lines may be added to your application's `.htaccess` file:

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session. This is primarily helpful if you choose to use HTTP Authentication to authenticate requests to your application's API. To accomplish this, [define a middleware](#) that calls the `onceBasic` method. If no response is returned by the `onceBasic` method, the request may be passed further into the application:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Symfony\Component\HttpFoundation\Response;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        return Auth::onceBasic() ?: $next($request);
    }
}
```

Next, attach the middleware to a route:

```
Route::get('/api/user', function () {
    // Only authenticated users may access this route...
})->middleware(AuthenticateOnceWithBasicAuth::class);
```

Logging Out

To manually log users out of your application, you may use the `logout` method provided by the `Auth` facade. This will remove the authentication information from the user's session so that subsequent requests are not authenticated.

In addition to calling the `logout` method, it is recommended that you invalidate the user's session and regenerate their [CSRF token](#). After logging the user out, you would typically redirect the user to the root of your application:

```

use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

/**
 * Log the user out of the application.
 */
public function logout(Request $request): RedirectResponse
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect('/');
}

```

Invalidating Sessions on Other Devices

Laravel also provides a mechanism for invalidating and "logging out" a user's sessions that are active on other devices without invalidating the session on their current device. This feature is typically utilized when a user is changing or updating their password and you would like to invalidate sessions on other devices while keeping the current device authenticated.

Before getting started, you should make sure that the `Illuminate\Session\Middleware\AuthenticateSession` middleware is included on the routes that should receive session authentication. Typically, you should place this middleware on a route group definition so that it can be applied to the majority of your application's routes. By default, the `AuthenticateSession` middleware may be attached to a route using the `auth.session` route middleware alias as defined in your application's HTTP kernel:

```

Route::middleware(['auth', 'auth.session'])->group(function () {
    Route::get('/', function () {
        // ...
    });
});

```

Then, you may use the `logoutOtherDevices` method provided by the `Auth` facade. This method requires the user to confirm their current password, which your application should accept through an input form:

```

use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($currentPassword);

```

When the `logoutOtherDevices` method is invoked, the user's other sessions will be invalidated entirely, meaning they will be "logged out" of all guards they were previously authenticated by.

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed or before the user is redirected to a sensitive area of the application. Laravel includes built-in middleware to make this process a breeze. Implementing this feature will require you to define two routes: one route to display a view asking the user to confirm their password and another route to confirm that the password is valid and redirect the user to their intended destination.



The following documentation discusses how to integrate with Laravel's password confirmation features directly; however, if you would like to get started more quickly, the [Laravel application starter kits](#) include support for this feature!

Configuration

After confirming their password, a user will not be asked to confirm their password again for three hours. However, you may configure the length of time before the user is re-prompted for their password by changing the value of the `password_timeout` configuration value within your application's `config/auth.php` configuration file.

Routing

The Password Confirmation Form

First, we will define a route to display a view that requests the user to confirm their password:

```
Route::get('/confirm-password', function () {
    return view('auth.confirm-password');
})->middleware('auth')->name('password.confirm');
```

As you might expect, the view that is returned by this route should have a form containing a `password` field. In addition, feel free to include text within the view that explains that the user is entering a protected area of the application and must confirm their password.

Confirming the Password

Next, we will define a route that will handle the form request from the "confirm password" view. This route will be responsible for validating the password and redirecting the user to their intended destination:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1']);
```

Before moving on, let's examine this route in more detail. First, the request's `password` field is determined to actually match the authenticated user's password. If the password is valid, we need to inform Laravel's session that the user has confirmed their password. The `passwordConfirmed` method will set a timestamp in the user's session that Laravel can use to determine when the user last confirmed their password. Finally, we can redirect the user to their intended destination.

Protecting Routes

You should ensure that any route that performs an action which requires recent password confirmation is assigned the `password.confirm` middleware. This middleware is included with the default installation of Laravel and will automatically store the user's intended destination in the session so that the user may be redirected to that location after confirming their password. After storing the user's intended destination in the session, the middleware will redirect the user to the `password.confirm` named route:

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

Adding Custom Guards

You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place your call to the `extend` method within a service provider. Since Laravel already ships with an `AuthServiceProvider`, we can place the code in that provider:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     */
    public function boot(): void
    {
        Auth::extend('jwt', function (Application $app, string $name, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\Guard...

            return new JwtGuard($app->createUserProvider($config['provider']));
        });
    }
}
```

As you can see in the example above, the callback passed to the `extend` method should return an implementation of `Illuminate\Contracts\Auth\Guard`. This interface contains a few methods you will need to implement to define a custom guard. Once your custom guard has been defined, you may reference the guard in the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

Closure Request Guards

The simplest way to implement a custom, HTTP request based authentication system is by using the `Auth::viaRequest` method. This method allows you to quickly define your authentication process using a single closure.

To get started, call the `Auth::viaRequest` method within the `boot` method of your `AuthServiceProvider`. The `viaRequest` method accepts an authentication driver name as its first argument. This name can be any string that describes your custom guard. The second argument passed to the method should be a closure that receives the incoming HTTP request and returns a user instance or, if authentication fails, `null`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Register any application authentication / authorization services.
 */
public function boot(): void
{
    Auth::viaRequest('custom-token', function (Request $request) {
        return User::where('token', (string) $request->token)->first();
    });
}
```

Once your custom authentication driver has been defined, you may configure it as a driver within the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

Finally, you may reference the guard when assigning the authentication middleware to a route:

```
Route::middleware('auth:api')->group(function () {
    // ...
});
```

Adding Custom User Providers

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication user provider. We will use the `provider` method on the `Auth` facade to define a custom user provider. The user provider resolver should return an implementation of `Illuminate\Contracts\Auth\UserProvider`:

```
<?php

namespace App\Providers;

use App\Extensions\MongoUserProvider;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     */
    public function boot(): void
    {
        Auth::provider('mongo', function (Application $app, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\UserProvider...

            return new MongoUserProvider($app->make('mongo.connection'));
        });
    }
}
```

After you have registered the provider using the `provider` method, you may switch to the new user provider in your `auth.php` configuration file. First, define a `provider` that uses your new driver:

```
'providers' => [
    'users' => [
        'driver' => 'mongo',
    ],
],
```

Finally, you may reference this provider in your `guards` configuration:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

The User Provider Contract

`Illuminate\Contracts\Auth\UserProvider` implementations are responsible for fetching an `Illuminate\Contracts\Auth\Authenticatable` implementation out of a persistent storage system, such as MySQL, MongoDB, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent the authenticated user:

Let's take a look at the `Illuminate\Contracts\Auth\UserProvider` contract:

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider
{
    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);
}
```

The `retrieveById` function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The `Authenticatable` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByToken` function retrieves a user by their unique `$identifier` and "remember me" `$token`, typically stored in a database column like `remember_token`. As with the previous method, the `Authenticatable` implementation with a matching token value should be returned by this method.

The `updateRememberToken` method updates the `$user` instance's `remember_token` with the new `$token`. A fresh token is assigned to users on a successful "remember me" authentication attempt or when the user is logging out.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to authenticate with an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition that searches for a user record with a "username" matching the value of `$credentials['username']`. The method should return an implementation of `Authenticatable`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given `$user` with the `$credentials` to authenticate the user. For example, this method will typically use the `Hash::check` method to compare the value of `$user->getAuthPassword()` to the value of `$credentials['password']`. This method should return `true` or `false` indicating whether the password is valid.

The Authenticatable Contract

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable` contract. Remember, user providers should return implementations of this interface from the `retrieveById`, `retrieveByToken`, and `retrieveByCredentials` methods:

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable
{
    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();
}
```

This interface is simple. The `getAuthIdentifierName` method should return the name of the "primary key" field of the user and the `getAuthIdentifier` method should return the "primary key" of the user. When using a MySQL back-end, this would likely be the auto-incrementing primary key assigned to the user record. The `getAuthPassword` method should return the user's hashed password.

This interface allows the authentication system to work with any "user" class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes an `App\Models\User` class in the `app/Models` directory which implements this interface.

Events

Laravel dispatches a variety of [events](#) during the authentication process. You may attach listeners to these events in your `EventServiceProvider`:

```

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [
        'App\Listeners\LogRegisteredUser',
    ],

    'Illuminate\Auth\Events\Attempting' => [
        'App\Listeners\LogAuthenticationAttempt',
    ],

    'Illuminate\Auth\Events\Authenticated' => [
        'App\Listeners\LogAuthenticated',
    ],

    'Illuminate\Auth\Events>Login' => [
        'App\Listeners\LogSuccessfulLogin',
    ],

    'Illuminate\Auth\Events\Failed' => [
        'App\Listeners\LogFailedLogin',
    ],

    'Illuminate\Auth\Events\Validated' => [
        'App\Listeners\LogValidated',
    ],

    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerified',
    ],

    'Illuminate\Auth\Events\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

    'Illuminate\Auth\Events\CurrentDeviceLogout' => [
        'App\Listeners\LogCurrentDeviceLogout',
    ],

    'Illuminate\Auth\Events\OtherDeviceLogout' => [
        'App\Listeners\LogOtherDeviceLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
    ],

    'Illuminate\Auth\Events>PasswordReset' => [
        'App\Listeners\LogPasswordReset',
    ],
];

```

Authorization

- [Introduction](#)
- [Gates](#)
 - [Writing Gates](#)
 - [Authorizing Actions](#)
 - [Gate Responses](#)
 - [Intercepting Gate Checks](#)
 - [Inline Authorization](#)
- [Creating Policies](#)
 - [Generating Policies](#)
 - [Registering Policies](#)
- [Writing Policies](#)
 - [Policy Methods](#)
 - [Policy Responses](#)
 - [Methods Without Models](#)
 - [Guest Users](#)
 - [Policy Filters](#)
- [Authorizing Actions Using Policies](#)
 - [Via the User Model](#)
 - [Via Controller Helpers](#)
 - [Via Middleware](#)
 - [Via Blade Templates](#)
 - [Supplying Additional Context](#)

Introduction

In addition to providing built-in [authentication](#) services, Laravel also provides a simple way to authorize user actions against a given resource. For example, even though a user is authenticated, they may not be authorized to update or delete certain Eloquent models or database records managed by your application. Laravel's authorization features provide an easy, organized way of managing these types of authorization checks.

Laravel provides two primary ways of authorizing actions: [gates](#) and [policies](#). Think of gates and policies like routes and controllers. Gates provide a simple, closure-based approach to authorization while policies, like controllers, group logic around a particular model or resource. In this documentation, we'll explore gates first and then examine policies.

You do not need to choose between exclusively using gates or exclusively using policies when building an application. Most applications will most likely contain some mixture of gates and policies, and that is perfectly fine! Gates are most applicable to actions that are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Gates

Writing Gates



Gates are a great way to learn the basics of Laravel's authorization features; however, when building robust Laravel applications you should consider using [policies](#) to organize your authorization rules.

Gates are simply closures that determine if a user is authorized to perform a given action. Typically, gates are defined within the `boot` method of the `App\Providers\AuthServiceProvider` class using the `Gate` facade. Gates always receive a user instance as their first argument and may optionally receive additional arguments such as a relevant Eloquent model.

In this example, we'll define a gate to determine if a user can update a given `App\Models\Post` model. The gate will accomplish this by comparing the user's `id` against the `user_id` of the user that created the post:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Gate::define('update-post', function (User $user, Post $post) {
        return $user->id === $post->user_id;
    });
}
```

Like controllers, gates may also be defined using a class callback array:

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

Authorizing Actions

To authorize an action using gates, you should use the `allows` or `denies` methods provided by the `Gate` facade. Note that you are not required to pass the currently authenticated user to these methods. Laravel will automatically take care of passing the user into the gate closure. It is typical to call the gate authorization methods within your application's controllers before performing an action that requires authorization:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Update the given post.
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if (! Gate::allows('update-post', $post)) {
            abort(403);
        }

        // Update the post...

        return redirect('/posts');
    }
}
```

If you would like to determine if a user other than the currently authenticated user is authorized to perform an action, you may use the `forUser` method on the `Gate` facade:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // The user can't update the post...
}
```

You may authorize multiple actions at a time using the `any` or `none` methods:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user can't update or delete the post...
}
```

Authorizing or Throwing Exceptions

If you would like to attempt to authorize an action and automatically throw an

`Illuminate\Auth\Access\AuthorizationException` if the user is not allowed to perform the given action, you may use the `Gate` facade's `authorize` method. Instances of `AuthorizationException` are automatically converted to a 403 HTTP response by Laravel's exception handler:

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

Supplying Additional Context

The gate methods for authorizing abilities (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) and the authorization [Blade directives](#) (`@can`, `@cannot`, `@canany`) can receive an array as their second argument. These array elements are passed as parameters to the gate closure, and can be used for additional context when making authorization decisions:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category, bool $pinned) {
    if (! $user->canPublishToGroup($category->group)) {
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false;
    }

    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
    // The user can create the post...
}
```

Gate Responses

So far, we have only examined gates that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an [Illuminate\Auth\Access\Response](#) from your gate:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::deny('You must be an administrator.');
});
```

Even when you return an authorization response from your gate, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('edit-settings');

// The action is authorized...
```

Customizing The HTTP Response Status

When an action is denied via a Gate, a `403` HTTP response is returned; however, it can sometimes be useful to return an alternative HTTP status code. You may customize the HTTP status code returned for a failed authorization check using the `denyWithStatus` static constructor on the `Illuminate\Auth\Access\Response` class:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::denyWithStatus(404);
});
```

Because hiding resources via a `404` response is such a common pattern for web applications, the `denyAsNotFound` method is offered for convenience:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::denyAsNotFound();
});
```

Intercepting Gate Checks

Sometimes, you may wish to grant all abilities to a specific user. You may use the `before` method to define a closure that is run before all other authorization checks:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::before(function (User $user, string $ability) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

If the `before` closure returns a non-null result that result will be considered the result of the authorization check.

You may use the `after` method to define a closure to be executed after all other authorization checks:

```
use App\Models\User;

Gate::after(function (User $user, string $ability, bool|null $result, mixed $arguments) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

Similar to the `before` method, if the `after` closure returns a non-null result that result will be considered the result of the authorization check.

Inline Authorization

Occasionally, you may wish to determine if the currently authenticated user is authorized to perform a given action without writing a dedicated gate that corresponds to the action. Laravel allows you to perform these types of "inline" authorization checks via the `Gate::allowIf` and `Gate::denyIf` methods. Inline authorization does not execute any defined "before" or "after" authorization hooks:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::allowIf(fn (User $user) => $user->isAdministrator());

Gate::denyIf(fn (User $user) => $user->banned());
```

If the action is not authorized or if no user is currently authenticated, Laravel will automatically throw an `Illuminate\Auth\Access\AuthorizationException` exception. Instances of `AuthorizationException` are automatically converted to a 403 HTTP response by Laravel's exception handler.

Creating Policies

Generating Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, if your application is a blog, you may have an `App\Models\Post` model and a corresponding `App\Policies\PostPolicy` to authorize user actions such as creating or updating posts.

You may generate a policy using the `make:policy` Artisan command. The generated policy will be placed in the `app/Policies` directory. If this directory does not exist in your application, Laravel will create it for you:

```
php artisan make:policy PostPolicy
```

The `make:policy` command will generate an empty policy class. If you would like to generate a class with example policy methods related to viewing, creating, updating, and deleting the resource, you may provide a `--model` option when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

Registering Policies

Once the policy class has been created, it needs to be registered. Registering policies is how we can inform Laravel which policy to use when authorizing actions against a given model type.

The `App\Providers\AuthServiceProvider` included with fresh Laravel applications contains a `policies` property which maps your Eloquent models to their corresponding policies. Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given Eloquent model:

```
<?php

namespace App\Providers;

use App\Models\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     */
    public function boot(): void
    {
        // ...
    }
}
```

Policy Auto-Discovery

Instead of manually registering model policies, Laravel can automatically discover policies as long as the model and policy follow standard Laravel naming conventions. Specifically, the policies must be in a `Policies` directory at or above the directory that contains your models. So, for example, the models may be placed in the `app/Models` directory while the policies may be placed in the `app/Policies` directory. In this situation, Laravel will check for policies in `app/Models/Policies` then `app/Policies`. In addition, the policy name must match the model name and have a `Policy` suffix. So, a `User` model would correspond to a `UserPolicy` policy class.

If you would like to define your own policy discovery logic, you may register a custom policy discovery callback using the `Gate::guessPolicyNamesUsing` method. Typically, this method should be called from the `boot` method of your application's `AuthServiceProvider`:

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function (string $modelClass) {
    // Return the name of the policy class for the given model...
});
```



Any policies that are explicitly mapped in your `AuthServiceProvider` will take precedence over any potentially auto-discovered policies.

Writing Policies

Policy Methods

Once the policy class has been registered, you may add methods for each action it authorizes. For example, let's define an `update` method on our `PostPolicy` which determines if a given `App\Models\User` can update a given `App\Models\Post` instance.

The `update` method will receive a `User` and a `Post` instance as its arguments, and should return `true` or `false` indicating whether the user is authorized to update the given `Post`. So, in this example, we will verify that the user's `id` matches the `user_id` on the post:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     */
    public function update(User $user, Post $post): bool
    {
        return $user->id === $post->user_id;
    }
}
```

You may continue to define additional methods on the policy as needed for the various actions it authorizes. For example, you might define `view` or `delete` methods to authorize various `Post` related actions, but remember you are free to give your policy methods any name you like.

If you used the `--model` option when generating your policy via the Artisan console, it will already contain methods for the `viewAny`, `view`, `create`, `update`, `delete`, `restore`, and `forceDelete` actions.



All policies are resolved via the Laravel [service container](#), allowing you to type-hint any needed dependencies in the policy's constructor to have them automatically injected.

Policy Responses

So far, we have only examined policy methods that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an `Illuminate\Auth\Access\Response` instance from your policy method:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
        ? Response::allow()
        : Response::deny('You do not own this post.');
}
```

When returning an authorization response from your policy, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
use Illuminate\Support\Facades\Gate;

$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('update', $post);

// The action is authorized...
```

Customizing the HTTP Response Status

When an action is denied via a policy method, a `403` HTTP response is returned; however, it can sometimes be useful to return an alternative HTTP status code. You may customize the HTTP status code returned for a failed authorization check using the `denyWithStatus` static constructor on the `Illuminate\Auth\Access\Response` class:

```

use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
        ? Response::allow()
        : Response::denyWithStatus(404);
}

```

Because hiding resources via a `404` response is such a common pattern for web applications, the `denyAsNotFound` method is offered for convenience:

```

use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
        ? Response::allow()
        : Response::denyAsNotFound();
}

```

Methods Without Models

Some policy methods only receive an instance of the currently authenticated user. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to determine if a user is authorized to create any posts at all. In these situations, your policy method should only expect to receive a user instance:

```

/**
 * Determine if the given user can create posts.
 */
public function create(User $user): bool
{
    return $user->role == 'writer';
}

```

Guest Users

By default, all gates and policies automatically return `false` if the incoming HTTP request was not initiated by an authenticated user. However, you may allow these authorization checks to pass through to your gates and policies by declaring an "optional" type-hint or supplying a `null` default value for the user argument definition:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     */
    public function update(?User $user, Post $post): bool
    {
        return $user?->id === $post->user_id;
    }
}
```

Policy Filters

For certain users, you may wish to authorize all actions within a given policy. To accomplish this, define a `before` method on the policy. The `before` method will be executed before any other methods on the policy, giving you an opportunity to authorize the action before the intended policy method is actually called. This feature is most commonly used for authorizing application administrators to perform any action:

```
use App\Models\User;

/**
 * Perform pre-authorization checks.
 */
public function before(User $user, string $ability): bool|null
{
    if ($user->isAdministrator()) {
        return true;
    }

    return null;
}
```

If you would like to deny all authorization checks for a particular type of user then you may return `false` from the `before` method. If `null` is returned, the authorization check will fall through to the policy method.



The `before` method of a policy class will not be called if the class doesn't contain a method with a name matching the name of the ability being checked.

Authorizing Actions Using Policies

Via the User Model

The `App\Models\User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cannot`. The `can` and `cannot` methods receive the name of the action you wish to authorize and

the relevant model. For example, let's determine if a user is authorized to update a given `App\Models\Post` model. Typically, this will be done within a controller method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Update the given post.
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }

        // Update the post...

        return redirect('/posts');
    }
}
```

If a [policy is registered](#) for the given model, the `can` method will automatically call the appropriate policy and return the boolean result. If no policy is registered for the model, the `can` method will attempt to call the closure-based Gate matching the given action name.

Actions That Don't Require Models

Remember, some actions may correspond to policy methods like `create` that do not require a model instance. In these situations, you may pass a class name to the `can` method. The class name will be used to determine which policy to use when authorizing the action:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Create a post.
     */
    public function store(Request $request): RedirectResponse
    {
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }

        // Create the post...

        return redirect('/posts');
    }
}
```

Via Controller Helpers

In addition to helpful methods provided to the `App\Models\User` model, Laravel provides a helpful `authorize` method to any of your controllers which extend the `App\Http\Controllers\Controller` base class.

Like the `can` method, this method accepts the name of the action you wish to authorize and the relevant model. If the action is not authorized, the `authorize` method will throw an `Illuminate\Auth\Access\AuthorizationException` exception which the Laravel exception handler will automatically convert to an HTTP response with a 403 status code:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        $this->authorize('update', $post);

        // The current user can update the blog post...

        return redirect('/posts');
    }
}
```

Actions That Don't Require Models

As previously discussed, some policy methods like `create` do not require a model instance. In these situations, you should pass a class name to the `authorize` method. The class name will be used to determine which policy to use when authorizing the action:

```
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

/**
 * Create a new blog post.
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request): RedirectResponse
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...

    return redirect('/posts');
}
```

Authorizing Resource Controllers

If you are utilizing [resource controllers](#), you may make use of the `authorizeResource` method in your controller's constructor. This method will attach the appropriate `can` middleware definitions to the resource controller's methods.

The `authorizeResource` method accepts the model's class name as its first argument, and the name of the route / request parameter that will contain the model's ID as its second argument. You should ensure your [resource controller](#) is created using the `--model` flag so that it has the required method signatures and type hints:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;

class PostController extends Controller
{
    /**
     * Create the controller instance.
     */
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

The following controller methods will be mapped to their corresponding policy method. When requests are routed to the given controller method, the corresponding policy method will automatically be invoked before the controller method is executed:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete



You may use the `make:policy` command with the `--model` option to quickly generate a policy class for a given model: `php artisan make:policy PostPolicy --model=Post`.

Via Middleware

Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers. By default, the `Illuminate\Auth\Middleware\Authorize` middleware is assigned the `can` key in your `App\Http\Kernel` class. Let's explore an example of using the `can` middleware to authorize that a user can update a post:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

In this example, we're passing the `can` middleware two arguments. The first is the name of the action we wish to authorize and the second is the route parameter we wish to pass to the policy method. In this case, since we are using implicit model binding, an `App\Models\Post` model will be passed to the policy method. If the user is not authorized to perform the given action, an HTTP response with a 403 status code will be returned by the middleware.

For convenience, you may also attach the `can` middleware to your route using the `can` method:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->can('update', 'post');
```

Actions That Don't Require Models

Again, some policy methods like `create` do not require a model instance. In these situations, you may pass a class name to the middleware. The class name will be used to determine which policy to use when authorizing the action:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Models\Post');
```

Specifying the entire class name within a string middleware definition can become cumbersome. For that reason, you may choose to attach the `can` middleware to your route using the `can` method:

```
use App\Models\Post;

Route::post('/post', function () {
    // The current user may create posts...
})->can('create', Post::class);
```

Via Blade Templates

When writing Blade templates, you may wish to display a portion of the page only if the user is authorized to perform a given action. For example, you may wish to show an update form for a blog post only if the user can actually update the post. In this situation, you may use the `@can` and `@cannot` directives:

```

@can('update', $post)
    <!-- The current user can update the post... -->
@elsecan('create', App\Models\Post::class)
    <!-- The current user can create new posts... -->
@endif
    <!-- ... -->
@endcan

@cannot('update', $post)
    <!-- The current user cannot update the post... -->
@elsecannot('create', App\Models\Post::class)
    <!-- The current user cannot create new posts... -->
@endcannot

```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above are equivalent to the following statements:

```

@if (Auth::user()->can('update', $post))
    <!-- The current user can update the post... -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- The current user cannot update the post... -->
@endunless

```

You may also determine if a user is authorized to perform any action from a given array of actions. To accomplish this, use the `@canany` directive:

```

@canany(['update', 'view', 'delete'], $post)
    <!-- The current user can update, view, or delete the post... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- The current user can create a post... -->
@endcanany

```

Actions That Don't Require Models

Like most of the other authorization methods, you may pass a class name to the `@can` and `@cannot` directives if the action does not require a model instance:

```

@can('create', App\Models\Post::class)
    <!-- The current user can create posts... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- The current user can't create posts... -->
@endcannot

```

Supplying Additional Context

When authorizing actions using policies, you may pass an array as the second argument to the various authorization functions and helpers. The first element in the array will be used to determine which policy should be invoked, while the rest of the array elements are passed as parameters to the policy method and can be used for additional context when making authorization decisions. For example, consider the following `PostPolicy` method definition which contains an additional `$category` parameter:

```
/**  
 * Determine if the given post can be updated by the user.  
 */  
public function update(User $user, Post $post, int $category): bool  
{  
    return $user->id === $post->user_id &&  
        $user->canUpdateCategory($category);  
}
```

When attempting to determine if the authenticated user can update a given post, we can invoke this policy method like so:

```
/**  
 * Update the given blog post.  
 *  
 * @throws \Illuminate\Auth\Access\AuthorizationException  
 */  
public function update(Request $request, Post $post): RedirectResponse  
{  
    $this->authorize('update', [$post, $request->category]);  
  
    // The current user can update the blog post...  
  
    return redirect('/posts');  
}
```

Email Verification

- [Introduction](#)
 - [Model Preparation](#)
 - [Database Preparation](#)
- [Routing](#)
 - [The Email Verification Notice](#)
 - [The Email Verification Handler](#)
 - [Resending the Verification Email](#)
 - [Protecting Routes](#)
- [Customization](#)
- [Events](#)

Introduction

Many web applications require users to verify their email addresses before using the application. Rather than forcing you to re-implement this feature by hand for each application you create, Laravel provides convenient built-in services for sending and verifying email verification requests.



Want to get started fast? Install one of the [Laravel application starter kits](#) in a fresh Laravel application. The starter kits will take care of scaffolding your entire authentication system, including email verification support.

Model Preparation

Before getting started, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\MustVerifyEmail` contract:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

Once this interface has been added to your model, newly registered users will automatically be sent an email containing an email verification link. As you can see by examining your application's `App\Providers\EventServiceProvider`, Laravel already contains a `SendEmailVerificationNotification` listener that is attached to the `Illuminate\Auth\Events\Registered` event. This event listener will send the email verification link to the user.

If you are manually implementing registration within your application instead of using a [starter kit](#), you should ensure that you are dispatching the `Illuminate\Auth\Events\Registered` event after a user's registration is successful:

```
use Illuminate\Auth\Events\Registered;
event(new Registered($user));
```

Database Preparation

Next, your `users` table must contain an `email_verified_at` column to store the date and time that the user's email address was verified. By default, the `users` table migration included with the Laravel framework already includes this column. So, all you need to do is run your database migrations:

```
php artisan migrate
```

Routing

To properly implement email verification, three routes will need to be defined. First, a route will be needed to display a notice to the user that they should click the email verification link in the verification email that Laravel sent them after registration.

Second, a route will be needed to handle requests generated when the user clicks the email verification link in the email.

Third, a route will be needed to resend a verification link if the user accidentally loses the first verification link.

The Email Verification Notice

As mentioned previously, a route should be defined that will return a view instructing the user to click the email verification link that was emailed to them by Laravel after registration. This view will be displayed to users when they try to access other parts of the application without verifying their email address first. Remember, the link is automatically emailed to the user as long as your `App\Models\User` model implements the `MustVerifyEmail` interface:

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

The route that returns the email verification notice should be named `verification.notice`. It is important that the route is assigned this exact name since the `verified` middleware included with Laravel will automatically redirect to this route name if a user has not verified their email address.



When manually implementing email verification, you are required to define the contents of the verification notice view yourself. If you would like scaffolding that includes all necessary authentication and verification views, check out the [Laravel application starter kits](#).

The Email Verification Handler

Next, we need to define a route that will handle requests generated when the user clicks the email verification link that was emailed to them. This route should be named `verification.verify` and be assigned the `auth` and `signed` middlewares:

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;

Route::get('/email/verify/{id}/{hash}', function (EmailVerificationRequest $request) {
    $request->fulfill();

    return redirect('/');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

Before moving on, let's take a closer look at this route. First, you'll notice we are using an `EmailVerificationRequest` request type instead of the typical `Illuminate\Http\Request` instance. The `EmailVerificationRequest` is a `form request` that is included with Laravel. This request will automatically take care of validating the request's `id` and `hash` parameters.

Next, we can proceed directly to calling the `fulfill` method on the request. This method will call the `markEmailAsVerified` method on the authenticated user and dispatch the `Illuminate\Auth\Events\Verified` event. The `markEmailAsVerified` method is available to the default `App\Models\User` model via the `Illuminate\Foundation\Auth\User` base class. Once the user's email address has been verified, you may redirect them wherever you wish.

Resending the Verification Email

Sometimes a user may misplace or accidentally delete the email address verification email. To accommodate this, you may wish to define a route to allow the user to request that the verification email be resent. You may then make a request to this route by placing a simple form submission button within your verification notice view:

```
use Illuminate\Http\Request;

Route::post('/email/verification-notification', function (Request $request) {
    $request->user()->sendEmailVerificationNotification();

    return back()->with('message', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
```

Protecting Routes

Route middleware may be used to only allow verified users to access a given route. Laravel ships with a `verified` middleware alias, which is an alias for the `Illuminate\Auth\Middleware\EnsureEmailIsVerified` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition. Typically, this middleware is paired with the `auth` middleware:

```
Route::get('/profile', function () {
    // Only verified users may access this route...
})->middleware(['auth', 'verified']);
```

If an unverified user attempts to access a route that has been assigned this middleware, they will automatically be redirected to the `verification.notice` named route.

Customization

Verification Email Customization

Although the default email verification notification should satisfy the requirements of most applications, Laravel allows you to customize how the email verification mail message is constructed.

To get started, pass a closure to the `toMailUsing` method provided by the `Illuminate\Auth\Notifications\VerifyEmail` notification. The closure will receive the notifiable model instance that is receiving the notification as well as the signed email verification URL that the user must visit to verify their email address. The closure should return an instance of `Illuminate\Notifications\Messages\MailMessage`. Typically, you should call the `toMailUsing` method from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    // ...

    VerifyEmail::toMailUsing(function (object $notifiable, string $url) {
        return (new MailMessage)
            ->subject('Verify Email Address')
            ->line('Click the button below to verify your email address.')
            ->action('Verify Email Address', $url);
    });
}
```



To learn more about mail notifications, please consult the [mail notification documentation](#).

Events

When using the [Laravel application starter kits](#), Laravel dispatches `events` during the email verification process. If you are manually handling email verification for your application, you may wish to manually dispatch these events after verification is completed. You may attach listeners to these events in your application's `EventServiceProvider`:

```
use App\Listeners\LogVerifiedUser;
use Illuminate\Auth\Events\Verified;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    Verified::class => [
        LogVerifiedUser::class,
    ],
];
```

Encryption

- [Introduction](#)
- [Configuration](#)
- [Using the Encrypter](#)

Introduction

Laravel's encryption services provide a simple, convenient interface for encrypting and decrypting text via OpenSSL using AES-256 and AES-128 encryption. All of Laravel's encrypted values are signed using a message authentication code (MAC) so that their underlying value can not be modified or tampered with once encrypted.

Configuration

Before using Laravel's encrypter, you must set the `key` configuration option in your `config/app.php` configuration file. This configuration value is driven by the `APP_KEY` environment variable. You should use the `php artisan key:generate` command to generate this variable's value since the `key:generate` command will use PHP's secure random bytes generator to build a cryptographically secure key for your application. Typically, the value of the `APP_KEY` environment variable will be generated for you during [Laravel's installation](#).

Using the Encrypter

Encrypting a Value

You may encrypt a value using the `encryptString` method provided by the `Crypt` facade. All encrypted values are encrypted using OpenSSL and the AES-256-CBC cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC). The integrated message authentication code will prevent the decryption of any values that have been tampered with by malicious users:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class DigitalOceanTokenController extends Controller
{
    /**
     * Store a DigitalOcean API token for the user.
     */
    public function store(Request $request): RedirectResponse
    {
        $request->user()->fill([
            'token' => Crypt::encryptString($request->token),
        ])->save();

        return redirect('/secrets');
    }
}
```

Decrypting a Value

You may decrypt values using the `decryptString` method provided by the `Crypt` facade. If the value can not be properly decrypted, such as when the message authentication code is invalid, an `Illuminate\Contracts\Encryption\DecryptException` will be thrown:

```
use Illuminate\Contracts\Encryption\DecryptException;
use Illuminate\Support\Facades\Crypt;

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

Hashing

- [Introduction](#)
- [Configuration](#)
- [Basic Usage](#)
 - [Hashing Passwords](#)
 - [Verifying That a Password Matches a Hash](#)
 - [Determining if a Password Needs to be Rehashed](#)

Introduction

The Laravel `Hash` facade provides secure Bcrypt and Argon2 hashing for storing user passwords. If you are using one of the [Laravel application starter kits](#), Bcrypt will be used for registration and authentication by default.

Bcrypt is a great choice for hashing passwords because its "work factor" is adjustable, which means that the time it takes to generate a hash can be increased as hardware power increases. When hashing passwords, slow is good. The longer an algorithm takes to hash a password, the longer it takes malicious users to generate "rainbow tables" of all possible string hash values that may be used in brute force attacks against applications.

Configuration

The default hashing driver for your application is configured in your application's `config/hashing.php` configuration file. There are currently several supported drivers: [Bcrypt](#) and [Argon2](#) (Argon2i and Argon2id variants).

Basic Usage

Hashing Passwords

You may hash a password by calling the `make` method on the `Hash` facade:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class PasswordController extends Controller
{
    /**
     * Update the password for the user.
     */
    public function update(Request $request): RedirectResponse
    {
        // Validate the new password length...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();

        return redirect('/profile');
    }
}
```

Adjusting The Bcrypt Work Factor

If you are using the Bcrypt algorithm, the `make` method allows you to manage the work factor of the algorithm using the `rounds` option; however, the default work factor managed by Laravel is acceptable for most applications:

```
$hashed = Hash::make('password', [
    'rounds' => 12,
]);
```

Adjusting The Argon2 Work Factor

If you are using the Argon2 algorithm, the `make` method allows you to manage the work factor of the algorithm using the `memory`, `time`, and `threads` options; however, the default values managed by Laravel are acceptable for most applications:

```
$hashed = Hash::make('password', [
    'memory' => 1024,
    'time' => 2,
    'threads' => 2,
]);
```



For more information on these options, please refer to the [official PHP documentation regarding Argon hashing](#).

Verifying That a Password Matches a Hash

The `check` method provided by the `Hash` facade allows you to verify that a given plain-text string corresponds to a given hash:

```
if (Hash::check('plain-text', $hashedPassword)) {  
    // The passwords match...  
}
```

Determining if a Password Needs to be Rehashed

The `needsRehash` method provided by the `Hash` facade allows you to determine if the work factor used by the hasher has changed since the password was hashed. Some applications choose to perform this check during the application's authentication process:

```
if (Hash::needsRehash($hashed)) {  
    $hashed = Hash::make('plain-text');  
}
```

Resetting Passwords

- [Introduction](#)
 - [Model Preparation](#)
 - [Database Preparation](#)
 - [Configuring Trusted Hosts](#)
- [Routing](#)
 - [Requesting the Password Reset Link](#)
 - [Resetting the Password](#)
- [Deleting Expired Tokens](#)
- [Customization](#)

Introduction

Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this by hand for every application you create, Laravel provides convenient services for sending password reset links and secure resetting passwords.



Want to get started fast? Install a Laravel [application starter kit](#) in a fresh Laravel application. Laravel's starter kits will take care of scaffolding your entire authentication system, including resetting forgotten passwords.

Model Preparation

Before using the password reset features of Laravel, your application's `App\Models\User` model must use the `Illuminate\Notifications\Notifiable` trait. Typically, this trait is already included on the default `App\Models\User` model that is created with new Laravel applications.

Next, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. The `App\Models\User` model included with the framework already implements this interface, and uses the `Illuminate\Auth\Passwords\CanResetPassword` trait to include the methods needed to implement the interface.

Database Preparation

A table must be created to store your application's password reset tokens. The migration for this table is included in the default Laravel application, so you only need to migrate your database to create this table:

```
php artisan migrate
```

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. This is particularly important when your application offers password reset functionality.

To learn more about this middleware, please consult the [TrustHosts middleware documentation](#).

Routing

To properly implement support for allowing users to reset their passwords, we will need to define several routes. First, we will need a pair of routes to handle allowing the user to request a password reset link via their email address. Second, we will need a pair of routes to handle actually resetting the password once the user visits the password reset link that is emailed to them and completes the password reset form.

Requesting the Password Reset Link

The Password Reset Link Request Form

First, we will define the routes that are needed to request password reset links. To get started, we will define a route that returns a view with the password reset link request form:

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

The view that is returned by this route should have a form containing an `email` field, which will allow the user to request a password reset link for a given email address.

Handling the Form Submission

Next, we will define a route that handles the form submission request from the "forgot password" view. This route will be responsible for validating the email address and sending the password reset request to the corresponding user:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

Route::post('/forgot-password', function (Request $request) {
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
        ? back()->with(['status' => __($status)])
        : back()->withErrors(['email' => __($status)]);
})->middleware('guest')->name('password.email');
```

Before moving on, let's examine this route in more detail. First, the request's `email` attribute is validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to send a password reset link to the user. The password broker will take care of retrieving the user by the given field (in this case, the email address) and sending the user a password reset link via Laravel's built-in [notification system](#).

The `sendResetLink` method returns a "status" slug. This status may be translated using Laravel's [localization](#) helpers in order to display a user-friendly message to the user regarding the status of their request. The translation of the password reset status is determined by your application's `lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file.



By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

You may be wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `sendResetLink` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the [authentication documentation](#).



When manually implementing password resets, you are required to define the contents of the views and routes yourself. If you would like scaffolding that includes all necessary authentication and verification logic, check out the [Laravel application starter kits](#).

Resetting the Password

The Password Reset Form

Next, we will define the routes necessary to actually reset the password once the user clicks on the password reset link that has been emailed to them and provides a new password. First, let's define the route that will display the reset password form that is displayed when the user clicks the reset password link. This route will receive a `token` parameter that we will use later to verify the password reset request:

```
Route::get('/reset-password/{token}', function (string $token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

The view that is returned by this route should display a form containing an `email` field, a `password` field, a `password_confirmation` field, and a hidden `token` field, which should contain the value of the secret `$token` received by our route.

Handling the Form Submission

Of course, we need to define a route to actually handle the password reset form submission. This route will be responsible for validating the incoming request and updating the user's password in the database:

```

use App\Models\User;
use Illuminate\Auth\Events\PasswordReset;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;

Route::post('/reset-password', function (Request $request) {
    $request->validate([
        'token' => 'required',
        'email' => 'required|email',
        'password' => 'required|min:8|confirmed',
    ]);

    $status = Password::reset(
        $request->only('email', 'password', 'password_confirmation', 'token'),
        function (User $user, string $password) {
            $user->forceFill([
                'password' => Hash::make($password)
            ])->setRememberToken(Str::random(60));

            $user->save();

            event(new PasswordReset($user));
        }
    );

    return $status === Password::PASSWORD_RESET
        ? redirect()->route('login')->with('status', __($status))
        : back()->withErrors(['email' => [__([$status])]]);
})->middleware('guest')->name('password.update');

```

Before moving on, let's examine this route in more detail. First, the request's `token`, `email`, and `password` attributes are validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to validate the password reset request credentials.

If the token, email address, and password given to the password broker are valid, the closure passed to the `reset` method will be invoked. Within this closure, which receives the user instance and the plain-text password provided to the password reset form, we may update the user's password in the database.

The `reset` method returns a "status" slug. This status may be translated using Laravel's [localization](#) helpers in order to display a user-friendly message to the user regarding the status of their request. The translation of the password reset status is determined by your application's `lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file. If your application does not contain a `lang` directory, you may create it using the `lang:publish` Artisan command.

Before moving on, you may be wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `reset` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the [authentication documentation](#).

Deleting Expired Tokens

Password reset tokens that have expired will still be present within your database. However, you may easily delete these records using the `auth:clear-resets` Artisan command:

```
php artisan auth:clear-resets
```

If you would like to automate this process, consider adding the command to your application's [scheduler](#):

```
$schedule->command('auth:clear-resets')->everyFifteenMinutes();
```

Customization

Reset Link Customization

You may customize the password reset link URL using the `createUrlUsing` method provided by the `ResetPassword` notification class. This method accepts a closure which receives the user instance that is receiving the notification as well as the password reset link token. Typically, you should call this method from your `App\Providers\AuthServiceProvider` service provider's `boot` method:

```
use App\Models\User;
use Illuminate\Auth\Notifications\ResetPassword;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    ResetPassword::createUrlUsing(function (User $user, string $token) {
        return 'https://example.com/reset-password?token='.$token;
    });
}
```

Reset Email Customization

You may easily modify the notification class used to send the password reset link to the user. To get started, override the `sendPasswordResetNotification` method on your `App\Models\User` model. Within this method, you may send the notification using any [notification class](#) of your own creation. The password reset `$token` is the first argument received by the method. You may use this `$token` to build the password reset URL of your choice and send your notification to the user:

```
use App\Notifications\ResetPasswordNotification;

/**
 * Send a password reset notification to the user.
 *
 * @param string $token
 */
public function sendPasswordResetNotification($token): void
{
    $url = 'https://example.com/reset-password?token='.$token;

    $this->notify(new ResetPasswordNotification($url));
}
```

Chapter 7

Database

Database: Getting Started

- [Introduction](#)
 - [Configuration](#)
 - [Read and Write Connections](#)
- [Running SQL Queries](#)
 - [Using Multiple Database Connections](#)
 - [Listening for Query Events](#)
 - [Monitoring Cumulative Query Time](#)
- [Database Transactions](#)
- [Connecting to the Database CLI](#)
- [Inspecting Your Databases](#)
- [Monitoring Your Databases](#)

Introduction

Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a [fluent query builder](#), and the [Eloquent ORM](#). Currently, Laravel provides first-party support for five databases:

- MariaDB 10.10+ ([Version Policy](#))
- MySQL 5.7+ ([Version Policy](#))
- PostgreSQL 11.0+ ([Version Policy](#))
- SQLite 3.8.8+
- SQL Server 2017+ ([Version Policy](#))

Configuration

The configuration for Laravel's database services is located in your application's `config/database.php` configuration file. In this file, you may define all of your database connections, as well as specify which connection should be used by default. Most of the configuration options within this file are driven by the values of your application's environment variables. Examples for most of Laravel's supported database systems are provided in this file.

By default, Laravel's sample [environment configuration](#) is ready to use with [Laravel Sail](#), which is a Docker configuration for developing Laravel applications on your local machine. However, you are free to modify your database configuration as needed for your local database.

SQLite Configuration

SQLite databases are contained within a single file on your filesystem. You can create a new SQLite database using the `touch` command in your terminal: `touch database/database.sqlite`. After the database has been created, you may easily configure your environment variables to point to this database by placing the absolute path to the database in the `DB_DATABASE` environment variable:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

To enable foreign key constraints for SQLite connections, you should set the `DB_FOREIGN_KEYS` environment variable to `true`:

```
DB_FOREIGN_KEYS=true
```

Microsoft SQL Server Configuration

To use a Microsoft SQL Server database, you should ensure that you have the `sqlsrv` and `pdo_sqlsrv` PHP extensions installed as well as any dependencies they may require such as the Microsoft SQL ODBC driver.

Configuration Using URLs

Typically, database connections are configured using multiple configuration values such as `host`, `database`, `username`, `password`, etc. Each of these configuration values has its own corresponding environment variable. This means that when configuring your database connection information on a production server, you need to manage several environment variables.

Some managed database providers such as AWS and Heroku provide a single database "URL" that contains all of the connection information for the database in a single string. An example database URL may look something like the following:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

These URLs typically follow a standard schema convention:

```
driver://username:password@host:port/database?options
```

For convenience, Laravel supports these URLs as an alternative to configuring your database with multiple configuration options. If the `url` (or corresponding `DATABASE_URL` environment variable) configuration option is present, it will be used to extract the database connection and credential information.

Read and Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

Note that three keys have been added to the configuration array: `read`, `write` and `sticky`. The `read` and `write` keys have array values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` configuration array.

You only need to place items in the `read` and `write` arrays if you wish to override the values from the main `mysql` array. So, in this case, `192.168.1.1` will be used as the host for the "read" connection, while `192.168.1.3` will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` array will be shared across both connections. When multiple values exist in the `host` configuration array, a database host will be randomly chosen for each request.

The `sticky` Option

The `sticky` option is an *optional* value that can be used to allow the immediate reading of records that have been written to the database during the current request cycle. If the `sticky` option is enabled and a "write" operation has been performed against the database during the current request cycle, any further "read" operations will use the "write" connection. This ensures that any data written during the request cycle can be immediately read back from the database during that same request. It is up to you to decide if this is the desired behavior for your application.

Running SQL Queries

Once you have configured your database connection, you may run queries using the `DB` facade. The `DB` facade provides methods for each type of query: `select`, `update`, `insert`, `delete`, and `statement`.

Running a Select Query

To run a basic SELECT query, you may use the `select` method on the `DB` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     */
    public function index(): View
    {
        $users = DB::select('select * from users where active = ?', [1]);

        return view('user.index', ['users' => $users]);
    }
}
```

The first argument passed to the `select` method is the SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the `where` clause constraints. Parameter binding provides protection against SQL injection.

The `select` method will always return an `array` of results. Each result within the array will be a PHP `stdClass` object representing a record from the database:

```
use Illuminate\Support\Facades\DB;

$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}
```

Selecting Scalar Values

Sometimes your database query may result in a single, scalar value. Instead of being required to retrieve the query's scalar result from a record object, Laravel allows you to retrieve this value directly using the `scalar` method:

```
$burgers = DB::scalar(
    "select count(case when food = 'burger' then 1 end) as burgers from menu"
);
```

Selecting Multiple Result Sets

If your application calls stored procedures that return multiple result sets, you may use the `selectResultSets` method to retrieve all of the result sets returned by the stored procedure:

```
[$options, $notifications] = DB::selectResultSets(
    "CALL get_user_options_and_notifications(?)", $request->user()->id
);
```

Using Named Bindings

Instead of using `?` to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Running an Insert Statement

To execute an `insert` statement, you may use the `insert` method on the `DB` facade. Like `select`, this method accepts the SQL query as its first argument and bindings as its second argument:

```
use Illuminate\Support\Facades\DB;

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

Running an Update Statement

The `update` method should be used to update existing records in the database. The number of rows affected by the statement is returned by the method:

```
use Illuminate\Support\Facades\DB;

$affected = DB::update(
    'update users set votes = 100 where name = ?',
    ['Anita']
);
```

Running a Delete Statement

The `delete` method should be used to delete records from the database. Like `update`, the number of rows affected will be returned by the method:

```
use Illuminate\Support\Facades\DB;

$deleted = DB::delete('delete from users');
```

Running a General Statement

Some database statements do not return any value. For these types of operations, you may use the `statement` method on the `DB` facade:

```
DB::statement('drop table users');
```

Running an Unprepared Statement

Sometimes you may want to execute an SQL statement without binding any values. You may use the `DB` facade's `unprepared` method to accomplish this:

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```



Since unprepared statements do not bind parameters, they may be vulnerable to SQL injection. You should never allow user controlled values within an unprepared statement.

Implicit Commits

When using the `DB` facade's `statement` and `unprepared` methods within transactions you must be careful to avoid statements that cause [implicit commits](#). These statements will cause the database engine to indirectly commit the entire transaction, leaving Laravel unaware of the database's transaction level. An example of such a statement is creating a database table:

```
DB::unprepared('create table a (col varchar(1) null');
```

Please refer to the MySQL manual for [a list of all statements](#) that trigger implicit commits.

Using Multiple Database Connections

If your application defines multiple connections in your `config/database.php` configuration file, you may access each connection via the `connection` method provided by the `DB` facade. The connection name passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file or configured at runtime using the `config` helper:

```
use Illuminate\Support\Facades\DB;

$users = DB::connection('sqlite')->select(/* ... */);
```

You may access the raw, underlying PDO instance of a connection using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Listening for Query Events

If you would like to specify a closure that is invoked for each SQL query executed by your application, you may use the `DB` facade's `listen` method. This method can be useful for logging queries or debugging. You may register your query listener closure in the `boot` method of a [service provider](#):

```
<?php

namespace App\Providers;

use Illuminate\Database\Events\QueryExecuted;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        DB::listen(function (QueryExecuted $query) {
            // $query->sql;
            // $query->bindings;
            // $query->time;
        });
    }
}
```

Monitoring Cumulative Query Time

A common performance bottleneck of modern web applications is the amount of time they spend querying databases. Thankfully, Laravel can invoke a closure or callback of your choice when it spends too much time querying the database during a single request. To get started, provide a query time threshold (in milliseconds) and closure to the `whenQueryingForLongerThan` method. You may invoke this method in the `boot` method of a [service provider](#):

```
<?php

namespace App\Providers;

use Illuminate\Database\Connection;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
use Illuminate\Database\Events\QueryExecuted;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        DB::whenQueryingForLongerThan(500, function (Connection $connection, QueryExecuted $event) {
            // Notify development team...
        });
    }
}
```

Database Transactions

You may use the `transaction` method provided by the `DB` facade to run a set of operations within a database transaction. If an exception is thrown within the transaction closure, the transaction will automatically be rolled back and the exception is re-thrown. If the closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the `transaction` method:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
});
```

Handling Deadlocks

The `transaction` method accepts an optional second argument which defines the number of times a transaction should be retried when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the `beginTransaction` method provided by the `DB` facade:

```
use Illuminate\Support\Facades\DB;

DB::beginTransaction();
```

You can rollback the transaction via the `rollBack` method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the `commit` method:

```
DB::commit();
```



The `DB` facade's transaction methods control the transactions for both the [query builder](#) and [Eloquent ORM](#).

Connecting to the Database CLI

If you would like to connect to your database's CLI, you may use the `db` Artisan command:

```
php artisan db
```

If needed, you may specify a database connection name to connect to a database connection that is not the default connection:

```
php artisan db mysql
```

Inspecting Your Databases

Using the `db:show` and `db:table` Artisan commands, you can get valuable insight into your database and its associated tables. To see an overview of your database, including its size, type, number of open connections, and a summary of its tables, you may use the `db:show` command:

```
php artisan db:show
```

You may specify which database connection should be inspected by providing the database connection name to the command via the `--database` option:

```
php artisan db:show --database=pgsql
```

If you would like to include table row counts and database view details within the output of the command, you may provide the `--counts` and `--views` options, respectively. On large databases, retrieving row counts and view details can be slow:

```
php artisan db:show --counts --views
```

Table Overview

If you would like to get an overview of an individual table within your database, you may execute the `db:table` Artisan command. This command provides a general overview of a database table, including its columns, types, attributes, keys, and indexes:

```
php artisan db:table users
```

Monitoring Your Databases

Using the `db:monitor` Artisan command, you can instruct Laravel to dispatch an `Illuminate\Database\Events\DatabaseBusy` event if your database is managing more than a specified number of open connections.

To get started, you should schedule the `db:monitor` command to run every minute. The command accepts the names of the database connection configurations that you wish to monitor as well as the maximum number of open connections that should be tolerated before dispatching an event:

```
php artisan db:monitor --databases=mysql,pgsql --max=100
```

Scheduling this command alone is not enough to trigger a notification alerting you of the number of open connections. When the command encounters a database that has an open connection count that exceeds your threshold, a `DatabaseBusy` event will be dispatched. You should listen for this event within your application's `EventServiceProvider` in order to send a notification to you or your development team:

```
use App\Notifications\DatabaseApproachingMaxConnections;
use Illuminate\Database\Events\DatabaseBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * Register any other events for your application.
 */
public function boot(): void
{
    Event::listen(function (DatabaseBusy $event) {
        Notification::route('mail', 'dev@example.com')
            ->notify(new DatabaseApproachingMaxConnections(
                $event->connectionName,
                $event->connections
            ));
    });
}
```

Database: Query Builder

- [Introduction](#)
- [Running Database Queries](#)
 - [Chunking Results](#)
 - [Streaming Results Lazily](#)
 - [Aggregates](#)
- [Select Statements](#)
- [Raw Expressions](#)
- [Joins](#)
- [Unions](#)
- [Basic Where Clauses](#)
 - [Where Clauses](#)
 - [Or Where Clauses](#)
 - [Where Not Clauses](#)
 - [Where Any / All Clauses](#)
 - [JSON Where Clauses](#)
 - [Additional Where Clauses](#)
 - [Logical Grouping](#)
- [Advanced Where Clauses](#)
 - [Where Exists Clauses](#)
 - [Subquery Where Clauses](#)
 - [Full Text Where Clauses](#)
- [Ordering, Grouping, Limit and Offset](#)
 - [Ordering](#)
 - [Grouping](#)
 - [Limit and Offset](#)
- [Conditional Clauses](#)
- [Insert Statements](#)
 - [Upserts](#)
- [Update Statements](#)
 - [Updating JSON Columns](#)
 - [Increment and Decrement](#)
- [Delete Statements](#)
- [Pessimistic Locking](#)
- [Debugging](#)

Introduction

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works perfectly with all of Laravel's supported database systems.

The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings.



PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns.

Running Database Queries

Retrieving All Rows From a Table

You may use the `table` method provided by the `DB` facade to begin a query. The `table` method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally retrieve the results of the query using the `get` method:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     */
    public function index(): View
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

The `get` method returns an `Illuminate\Support\Collection` instance containing the results of the query where each result is an instance of the PHP `stdClass` object. You may access each column's value by accessing the column as a property of the object:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```



Laravel collections provide a variety of extremely powerful methods for mapping and reducing data. For more information on Laravel collections, check out the [collection documentation](#).

Retrieving a Single Row / Column From a Table

If you just need to retrieve a single row from a database table, you may use the `DB` facade's `first` method. This method will return a single `stdClass` object:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

If you don't need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

To retrieve a single row by its `id` column value, use the `find` method:

```
$user = DB::table('users')->find(3);
```

Retrieving a List of Column Values

If you would like to retrieve an `Illuminate\Support\Collection` instance containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve a collection of user titles:

```
use Illuminate\Support\Facades\DB;

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may specify the column that the resulting collection should use as its keys by providing a second argument to the `pluck` method:

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

Chunking Results

If you need to work with thousands of database records, consider using the `chunk` method provided by the `DB` facade. This method retrieves a small chunk of results at a time and feeds each chunk into a closure for processing. For example, let's retrieve the entire `users` table in chunks of 100 records at a time:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    foreach ($users as $user) {
        // ...
    }
});
```

You may stop further chunks from being processed by returning `false` from the closure:

```
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    // Process the records...

    return false;
});
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. If you plan to update the retrieved records while chunking, it is always best to use the `chunkById` method instead. This method will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
->chunkById(100, function (Collection $users) {
    foreach ($users as $user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    }
});
```



When updating or deleting records inside the chunk callback, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the chunked results.

Streaming Results Lazily

The `lazy` method works similarly to the `chunk` method in the sense that it executes the query in chunks. However, instead of passing each chunk into a callback, the `lazy()` method returns a [LazyCollection](#), which lets you interact with the results as a single stream:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function (object $user) {
    // ...
});
```

Once again, if you plan to update the retrieved records while iterating over them, it is best to use the `lazyById` or `lazyByIdDesc` methods instead. These methods will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
->lazyById()->each(function (object $user) {
    DB::table('users')
        ->where('id', $user->id)
        ->update(['active' => true]);
});
```



When updating or deleting records while iterating over them, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the results.

Aggregates

The query builder also provides a variety of methods for retrieving aggregate values like `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses to fine-tune how your aggregate value is calculated:

```
$price = DB::table('orders')
    ->where('finalized', 1)
    ->avg('price');
```

Determining if Records Exist

Instead of using the `count` method to determine if any records exist that match your query's constraints, you may use the `exists` and `doesntExist` methods:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

Select Statements

Specifying a Select Clause

You may not always want to select all columns from a database table. Using the `select` method, you can specify a custom "select" clause for the query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
    ->select('name', 'email as user_email')
    ->get();
```

The `distinct` method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the `addSelect` method:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Raw Expressions

Sometimes you may need to insert an arbitrary string into a query. To create a raw string expression, you may use the `raw` method provided by the `DB` facade:

```
$users = DB::table('users')
    ->select(DB::raw('count(*) as user_count, status'))
    ->where('status', '<>', 1)
    ->groupBy('status')
    ->get();
```



Raw statements will be injected into the query as strings, so you should be extremely careful to avoid creating SQL injection vulnerabilities.

Raw Methods

Instead of using the `DB::raw` method, you may also use the following methods to insert a raw expression into various parts of your query. **Remember, Laravel can not guarantee that any query using raw expressions is protected against SQL injection vulnerabilities.**

`selectRaw`

The `selectRaw` method can be used in place of `addSelect(DB::raw(/* ... */))`. This method accepts an optional array of bindings as its second argument:

```
$orders = DB::table('orders')
    ->selectRaw('price * ? as price_with_tax', [1.0825])
    ->get();
```

`whereRaw / orWhereRaw`

The `whereRaw` and `orWhereRaw` methods can be used to inject a raw "where" clause into your query. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
    ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
    ->get();
```

`havingRaw / orHavingRaw`

The `havingRaw` and `orHavingRaw` methods may be used to provide a raw string as the value of the "having" clause. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
    ->select('department', DB::raw('SUM(price) as total_sales'))
    ->groupBy('department')
    ->havingRaw('SUM(price) > ?', [2500])
    ->get();
```

`orderByRaw`

The `orderByRaw` method may be used to provide a raw string as the value of the "order by" clause:

```
$orders = DB::table('orders')
    ->orderByRaw('updated_at - created_at DESC')
    ->get();
```

groupByRaw

The `groupByRaw` method may be used to provide a raw string as the value of the `group by` clause:

```
$orders = DB::table('orders')
    ->select('city', 'state')
    ->groupByRaw('city, state')
    ->get();
```

Joins

Inner Join Clause

The query builder may also be used to add join clauses to your queries. To perform a basic "inner join", you may use the `join` method on a query builder instance. The first argument passed to the `join` method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. You may even join multiple tables in a single query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

Left Join / Right Join Clause

If you would like to perform a "left join" or "right join" instead of an "inner join", use the `leftJoin` or `rightJoin` methods. These methods have the same signature as the `join` method:

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();

$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

Cross Join Clause

You may use the `crossJoin` method to perform a "cross join". Cross joins generate a cartesian product between the first table and the joined table:

```
$sizes = DB::table('sizes')
    ->crossJoin('colors')
    ->get();
```

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a closure as the second argument to the `join` method. The closure will receive a `Illuminate\Database\Query\JoinClause` instance which allows you to specify constraints on the "join" clause:

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */);
    })
    ->get();
```

If you would like to use a "where" clause on your joins, you may use the `where` and `orWhere` methods provided by the `JoinClause` instance. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.user_id', '>', 5);
    })
    ->get();
```

Subquery Joins

You may use the `joinSub`, `leftJoinSub`, and `rightJoinSub` methods to join a query to a subquery. Each of these methods receives three arguments: the subquery, its table alias, and a closure that defines the related columns. In this example, we will retrieve a collection of users where each user record also contains the `created_at` timestamp of the user's most recently published blog post:

```
$latestPosts = DB::table('posts')
    ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
    ->where('is_published', true)
    ->groupBy('user_id');

$users = DB::table('users')
    ->joinSub($latestPosts, 'latest_posts', function (JoinClause $join) {
        $join->on('users.id', '=', 'latest_posts.user_id');
    })->get();
```

Lateral Joins



Lateral joins are currently supported by PostgreSQL, MySQL >= 8.0.14, and SQL Server.

You may use the `joinLateral` and `leftJoinLateral` methods to perform a "lateral join" with a subquery. Each of these methods receives two arguments: the subquery and its table alias. The join condition(s) should be specified within the `where` clause of the given subquery. Lateral joins are evaluated for each row and can reference columns outside the subquery.

In this example, we will retrieve a collection of users as well as the user's three most recent blog posts. Each user can produce up to three rows in the result set: one for each of their most recent blog posts. The join condition is specified with a `whereColumn` clause within the subquery, referencing the current user row:

```
$latestPosts = DB::table('posts')
    ->select('id as post_id', 'title as post_title', 'created_at as
post_created_at')
    ->whereColumn('user_id', 'users.id')
    ->orderBy('created_at', 'desc')
    ->limit(3);

$users = DB::table('users')
    ->joinLateral($latestPosts, 'latest_posts')
    ->get();
```

Unions

The query builder also provides a convenient method to "union" two or more queries together. For example, you may create an initial query and use the `union` method to union it with more queries:

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
    ->whereNull('first_name');

$users = DB::table('users')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

In addition to the `union` method, the query builder provides a `unionAll` method. Queries that are combined using the `unionAll` method will not have their duplicate results removed. The `unionAll` method has the same method signature as the `union` method.

Basic Where Clauses

Where Clauses

You may use the query builder's `where` method to add "where" clauses to the query. The most basic call to the `where` method requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to compare against the column's value.

For example, the following query retrieves users where the value of the `votes` column is equal to `100` and the value of the `age` column is greater than `35`:

```
$users = DB::table('users')
    ->where('votes', '=', 100)
    ->where('age', '>', 35)
    ->get();
```

For convenience, if you want to verify that a column is `=` to a given value, you may pass the value as the second argument to the `where` method. Laravel will assume you would like to use the `=` operator:

```
$users = DB::table('users')->where('votes', 100)->get();
```

As previously mentioned, you may use any operator that is supported by your database system:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->get();

$users = DB::table('users')
    ->where('votes', '<>', 100)
    ->get();

$users = DB::table('users')
    ->where('name', 'like', 'T%')
    ->get();
```

You may also pass an array of conditions to the `where` function. Each element of the array should be an array containing the three arguments typically passed to the `where` method:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```



PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns.

Or Where Clauses

When chaining together calls to the query builder's `where` method, the "where" clauses will be joined together using the `and` operator. However, you may use the `orWhere` method to join a clause to the query using the `or` operator. The `orWhere` method accepts the same arguments as the `where` method:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

If you need to group an "or" condition within parentheses, you may pass a closure as the first argument to the `orWhere` method:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere(function (Builder $query) {
        $query->where('name', 'Abigail')
            ->where('votes', '>', 50);
    })
    ->get();
```

The example above will produce the following SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```



You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

Where Not Clauses

The `whereNot` and `orWhereNot` methods may be used to negate a given group of query constraints. For example, the following query excludes products that are on clearance or which have a price that is less than ten:

```
$products = DB::table('products')
    ->whereNot(function (Builder $query) {
        $query->where('clearance', true)
            ->orWhere('price', '<', 10);
    })
    ->get();
```

Where Any / All Clauses

Sometimes you may need to apply the same query constraints to multiple columns. For example, you may want to retrieve all records where any columns in a given list are `LIKE` a given value. You may accomplish this using the `whereAny` method:

```
$users = DB::table('users')
    ->where('active', true)
    ->whereAny([
        'name',
        'email',
        'phone',
    ], 'LIKE', 'Example%')
    ->get();
```

The query above will result in the following SQL:

```
SELECT *
FROM users
WHERE active = true AND (
    name LIKE 'Example%' OR
    email LIKE 'Example%' OR
    phone LIKE 'Example%'
)
```

Similarly, the `whereAll` method may be used to retrieve records where all of the given columns match a given constraint:

```
$posts = DB::table('posts')
    ->where('published', true)
    ->whereAll([
        'title',
        'content',
    ], 'LIKE', '%Laravel%')
    ->get();
```

The query above will result in the following SQL:

```
SELECT *
FROM posts
WHERE published = true AND (
    title LIKE '%Laravel%' AND
    content LIKE '%Laravel%'
)
```

JSON Where Clauses

Laravel also supports querying JSON column types on databases that provide support for JSON column types. Currently, this includes MySQL 5.7+, PostgreSQL, SQL Server 2016, and SQLite 3.39.0 (with the [JSON1 extension](#)). To query a JSON column, use the `where` operator:

```
$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
    ->get();
```

You may use `whereJsonContains` to query JSON arrays:

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', 'en')
    ->get();
```

If your application uses the MySQL or PostgreSQL databases, you may pass an array of values to the `whereJsonContains` method:

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', ['en', 'de'])
    ->get();
```

You may use `whereJsonLength` method to query JSON arrays by their length:

```
$users = DB::table('users')
    ->whereJsonLength('options->languages', 0)
    ->get();

$users = DB::table('users')
    ->whereJsonLength('options->languages', '>', 1)
    ->get();
```

Additional Where Clauses

`whereBetween / orWhereBetween`

The `whereBetween` method verifies that a column's value is between two values:

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])
    ->get();
```

`whereNotBetween / orWhereNotBetween`

The `whereNotBetween` method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
    ->whereNotBetween('votes', [1, 100])
    ->get();
```

whereBetweenColumns / whereNotBetweenColumns / orWhereBetweenColumns / orWhereNotBetweenColumns

The `whereBetweenColumns` method verifies that a column's value is between the two values of two columns in the same table row:

```
$patients = DB::table('patients')
    ->whereBetweenColumns('weight', ['minimum_allowed_weight',
    'maximum_allowed_weight'])
    ->get();
```

The `whereNotBetweenColumns` method verifies that a column's value lies outside the two values of two columns in the same table row:

```
$patients = DB::table('patients')
    ->whereNotBetweenColumns('weight', ['minimum_allowed_weight',
    'maximum_allowed_weight'])
    ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

The `whereNotIn` method verifies that the given column's value is not contained in the given array:

```
$users = DB::table('users')
    ->whereNotIn('id', [1, 2, 3])
    ->get();
```

You may also provide a query object as the `whereIn` method's second argument:

```
$activeUsers = DB::table('users')->select('id')->where('is_active', 1);

$users = DB::table('comments')
    ->whereIn('user_id', $activeUsers)
    ->get();
```

The example above will produce the following SQL:

```
select * from comments where user_id in (
    select id
    from users
    where is_active = 1
)
```



If you are adding a large array of integer bindings to your query, the `whereIntegerInRaw` or `whereIntegerNotInRaw` methods may be used to greatly reduce your memory usage.

`whereNull / whereNotNull / orWhereNull / orWhereNotNull`

The `whereNull` method verifies that the value of the given column is `NULL`:

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

The `whereNotNull` method verifies that the column's value is not `NULL`:

```
$users = DB::table('users')
    ->whereNotNull('updated_at')
    ->get();
```

`whereDate / whereMonth / whereDay / whereYear / whereTime`

The `whereDate` method may be used to compare a column's value against a date:

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-12-31')
    ->get();
```

The `whereMonth` method may be used to compare a column's value against a specific month:

```
$users = DB::table('users')
    ->whereMonth('created_at', '12')
    ->get();
```

The `whereDay` method may be used to compare a column's value against a specific day of the month:

```
$users = DB::table('users')
    ->whereDay('created_at', '31')
    ->get();
```

The `whereYear` method may be used to compare a column's value against a specific year:

```
$users = DB::table('users')
    ->whereYear('created_at', '2016')
    ->get();
```

The `whereTime` method may be used to compare a column's value against a specific time:

```
$users = DB::table('users')
    ->whereTime('created_at', '=', '11:20:45')
    ->get();
```

whereColumn / orWhereColumn

The `whereColumn` method may be used to verify that two columns are equal:

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name')
    ->get();
```

You may also pass a comparison operator to the `whereColumn` method:

```
$users = DB::table('users')
    ->whereColumn('updated_at', '>', 'created_at')
    ->get();
```

You may also pass an array of column comparisons to the `whereColumn` method. These conditions will be joined using the `and` operator:

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', '=', 'last_name'],
        ['updated_at', '>', 'created_at'],
    ])->get();
```

Logical Grouping

Sometimes you may need to group several "where" clauses within parentheses in order to achieve your query's desired logical grouping. In fact, you should generally always group calls to the `orWhere` method in parentheses in order to avoid unexpected query behavior. To accomplish this, you may pass a closure to the `where` method:

```
$users = DB::table('users')
    ->where('name', '=', 'John')
    ->where(function (Builder $query) {
        $query->where('votes', '>', 100)
            ->orWhere('title', '=', 'Admin');
    })
    ->get();
```

As you can see, passing a closure into the `where` method instructs the query builder to begin a constraint group. The closure will receive a query builder instance which you can use to set the constraints that should be contained within the parenthesis group. The example above will produce the following SQL:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```



You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

Advanced Where Clauses

Where Exists Clauses

The `whereExists` method allows you to write "where exists" SQL clauses. The `whereExists` method accepts a closure which will receive a query builder instance, allowing you to define the query that should be placed inside of the "exists" clause:

```
$users = DB::table('users')
    ->whereExists(function (Builder $query) {
        $query->select(DB::raw(1))
            ->from('orders')
            ->whereColumn('orders.user_id', 'users.id');
    })
    ->get();
```

Alternatively, you may provide a query object to the `whereExists` method instead of a closure:

```
$orders = DB::table('orders')
    ->select(DB::raw(1))
    ->whereColumn('orders.user_id', 'users.id');

$users = DB::table('users')
    ->whereExists($orders)
    ->get();
```

Both of the examples above will produce the following SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

Subquery Where Clauses

Sometimes you may need to construct a "where" clause that compares the results of a subquery to a given value. You may accomplish this by passing a closure and a value to the `where` method. For example, the following query will retrieve all users who have a recent "membership" of a given type;

```
use App\Models\User;
use Illuminate\Database\Query\Builder;

$users = User::where(function (Builder $query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Or, you may need to construct a "where" clause that compares a column to the results of a subquery. You may accomplish this by passing a column, operator, and closure to the `where` method. For example, the following query will retrieve all income records where the amount is less than average;

```
use App\Models\Income;
use Illuminate\Database\Query\Builder;

$incomes = Income::where('amount', '<', function (Builder $query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Full Text Where Clauses



Full text where clauses are currently supported by MySQL and PostgreSQL.

The `whereFullText` and `orWhereFullText` methods may be used to add full text "where" clauses to a query for columns that have [full text indexes](#). These methods will be transformed into the appropriate SQL for the underlying database system by Laravel. For example, a `MATCH AGAINST` clause will be generated for applications utilizing MySQL:

```
$users = DB::table('users')
    ->whereFullText('bio', 'web developer')
    ->get();
```

Ordering, Grouping, Limit and Offset

Ordering

The `orderBy` Method

The `orderBy` method allows you to sort the results of the query by a given column. The first argument accepted by the `orderBy` method should be the column you wish to sort by, while the second argument determines the direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->get();
```

To sort by multiple columns, you may simply invoke `orderBy` as many times as necessary:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->orderBy('email', 'asc')
    ->get();
```

The `latest` and `oldest` Methods

The `latest` and `oldest` methods allow you to easily order results by date. By default, the result will be ordered by the table's `created_at` column. Or, you may pass the column name that you wish to sort by:

```
$user = DB::table('users')
    ->latest()
    ->first();
```

Random Ordering

The `inRandomOrder` method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

```
$randomUser = DB::table('users')
    ->inRandomOrder()
    ->first();
```

Removing Existing Orderings

The `reorder` method removes all of the "order by" clauses that have previously been applied to the query:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

You may pass a column and direction when calling the `reorder` method in order to remove all existing "order by" clauses and apply an entirely new order to the query:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

Grouping

The `groupBy` and `having` Methods

As you might expect, the `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

You can use the `havingBetween` method to filter the results within a given range:

```
$report = DB::table('orders')
    ->selectRaw('count(id) as number_of_orders, customer_id')
    ->groupBy('customer_id')
    ->havingBetween('number_of_orders', [5, 15])
    ->get();
```

You may pass multiple arguments to the `groupBy` method to group by multiple columns:

```
$users = DB::table('users')
    ->groupBy('first_name', 'status')
    ->having('account_id', '>', 100)
    ->get();
```

To build more advanced `having` statements, see the `havingRaw` method.

Limit and Offset

The `skip` and `take` Methods

You may use the `skip` and `take` methods to limit the number of results returned from the query or to skip a given number of results in the query:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternatively, you may use the `limit` and `offset` methods. These methods are functionally equivalent to the `take` and `skip` methods, respectively:

```
$users = DB::table('users')
    ->offset(10)
    ->limit(5)
    ->get();
```

Conditional Clauses

Sometimes you may want certain query clauses to apply to a query based on another condition. For instance, you may only want to apply a `where` statement if a given input value is present on the incoming HTTP request. You may accomplish this using the `when` method:

```
$role = $request->string('role');

$users = DB::table('users')
    ->when($role, function (Builder $query, string $role) {
        $query->where('role_id', $role);
    })
    ->get();
```

The `when` method only executes the given closure when the first argument is `true`. If the first argument is `false`, the closure will not be executed. So, in the example above, the closure given to the `when` method will only be invoked if the `role` field is present on the incoming request and evaluates to `true`.

You may pass another closure as the third argument to the `when` method. This closure will only execute if the first argument evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default ordering of a query:

```
$sortByVotes = $request->boolean('sort_by_votes');

$users = DB::table('users')
    ->when($sortByVotes, function (Builder $query, bool $sortByVotes) {
        $query->orderBy('votes');
    }, function (Builder $query) {
        $query->orderBy('name');
    })
    ->get();
```

Insert Statements

The query builder also provides an `insert` method that may be used to insert records into the database table. The `insert` method accepts an array of column names and values:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

You may insert several records at once by passing an array of arrays. Each array represents a record that should be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

The `insertOrIgnore` method will ignore errors while inserting records into the database. When using this method, you should be aware that duplicate record errors will be ignored and other types of errors may also be ignored depending on the database engine. For example, `insertOrIgnore` will bypass MySQL's strict mode:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

The `insertUsing` method will insert new records into the table while using a subquery to determine the data that should be inserted:

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
)->where('updated_at', '<=', now()->subMonth()));
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```



When using PostgreSQL the `insertGetId` method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the column name as the second parameter to the `insertGetId` method.

Upserts

The `upsert` method will insert records that do not exist and update the records that already exist with new values that you may specify. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of columns that should be updated if a matching record already exists in the database:

```
DB::table('flights')->upsert(
    [
        ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
        ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
    ],
    ['departure', 'destination'],
    ['price']
);
```

In the example above, Laravel will attempt to insert two records. If a record already exists with the same `departure` and `destination` column values, Laravel will update that record's `price` column.



All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index. In addition, the MySQL database driver ignores the second argument of the `upsert` method and always uses the "primary" and "unique" indexes of the table to detect existing records.

Update Statements

In addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs indicating the columns to be updated. The `update` method returns the number of affected rows. You may constrain the `update` query using `where` clauses:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);
```

Update or Insert

Sometimes you may want to update an existing record in the database or create it if no matching record exists. In this scenario, the `updateOrInsert` method may be used. The `updateOrInsert` method accepts two arguments: an array of conditions by which to find the record, and an array of column and value pairs indicating the columns to be updated.

The `updateOrInsert` method will attempt to locate a matching database record using the first argument's column and value pairs. If the record exists, it will be updated with the values in the second argument. If the record can not be found, a new record will be inserted with the merged attributes of both arguments:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

Updating JSON Columns

When updating a JSON column, you should use `[>]` syntax to update the appropriate key in the JSON object. This operation is supported on MySQL 5.7+ and PostgreSQL 9.5+:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['options->enabled' => true]);
```

Increment and Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. Both of these methods accept at least one argument: the column to modify. A second argument may be provided to specify the amount by which the column should be incremented or decremented:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

If needed, you may also specify additional columns to update during the increment or decrement operation:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

In addition, you may increment or decrement multiple columns at once using the `incrementEach` and `decrementEach` methods:

```
DB::table('users')->incrementEach([
    'votes' => 5,
    'balance' => 100,
]);
```

Delete Statements

The query builder's `delete` method may be used to delete records from the table. The `delete` method returns the number of affected rows. You may constrain `delete` statements by adding "where" clauses before calling the `delete` method:

```
$deleted = DB::table('users')->delete();

$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

If you wish to truncate an entire table, which will remove all records from the table and reset the auto-incrementing ID to zero, you may use the `truncate` method:

```
DB::table('users')->truncate();
```

Table Truncation and PostgreSQL

When truncating a PostgreSQL database, the `CASCADE` behavior will be applied. This means that all foreign key related records in other tables will be deleted as well.

Pessimistic Locking

The query builder also includes a few functions to help you achieve "pessimistic locking" when executing your `select` statements. To execute a statement with a "shared lock", you may call the `sharedLock` method. A shared lock prevents the selected rows from being modified until your transaction is committed:

```
DB::table('users')
    ->where('votes', '>', 100)
    ->sharedLock()
    ->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the selected records from being modified or from being selected with another shared lock:

```
DB::table('users')
    ->where('votes', '>', 100)
    ->lockForUpdate()
    ->get();
```

Debugging

You may use the `dd` and `dump` methods while building a query to dump the current query bindings and SQL. The `dd` method will display the debug information and then stop executing the request. The `dump` method will display the debug information but allow the request to continue executing:

```
DB::table('users')->where('votes', '>', 100)->dd();
DB::table('users')->where('votes', '>', 100)->dump();
```

The `dumpRawSql` and `ddRawSql` methods may be invoked on a query to dump the query's SQL with all parameter bindings properly substituted:

```
DB::table('users')->where('votes', '>', 100)->dumpRawSql();
DB::table('users')->where('votes', '>', 100)->ddRawSql();
```

Database: Pagination

- [Introduction](#)
- [Basic Usage](#)
 - [Paginating Query Builder Results](#)
 - [Paginating Eloquent Results](#)
 - [Cursor Pagination](#)
 - [Manually Creating a Paginator](#)
 - [Customizing Pagination URLs](#)
- [Displaying Pagination Results](#)
 - [Adjusting the Pagination Link Window](#)
 - [Converting Results to JSON](#)
- [Customizing the Pagination View](#)
 - [Using Bootstrap](#)
- [Paginator and LengthAwarePaginator Instance Methods](#)
- [Cursor Paginator Instance Methods](#)

Introduction

In other frameworks, pagination can be very painful. We hope Laravel's approach to pagination will be a breath of fresh air. Laravel's paginator is integrated with the [query builder](#) and [Eloquent ORM](#) and provides convenient, easy-to-use pagination of database records with zero configuration.

By default, the HTML generated by the paginator is compatible with the [Tailwind CSS framework](#); however, Bootstrap pagination support is also available.

Tailwind JIT

If you are using Laravel's default Tailwind pagination views and the Tailwind JIT engine, you should ensure your application's `tailwind.config.js` file's `content` key references Laravel's pagination views so that their Tailwind classes are not purged:

```
content: [
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
    './vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.blade.php',
],

```

Basic Usage

Paginating Query Builder Results

There are several ways to paginate items. The simplest is by using the `paginate` method on the [query builder](#) or an [Eloquent query](#). The `paginate` method automatically takes care of setting the query's "limit" and "offset" based on the current page being viewed by the user. By default, the current page is detected by the value of the `page` query string argument on the HTTP request. This value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

In this example, the only argument passed to the `paginate` method is the number of items you would like displayed "per page". In this case, let's specify that we would like to display `15` items per page:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show all application users.
     */
    public function index(): View
    {
        return view('user.index', [
            'users' => DB::table('users')->paginate(15)
        ]);
    }
}
```

Simple Pagination

The `paginate` method counts the total number of records matched by the query before retrieving the records from the database. This is done so that the paginator knows how many pages of records there are in total. However, if you do not plan to show the total number of pages in your application's UI then the record count query is unnecessary.

Therefore, if you only need to display simple "Next" and "Previous" links in your application's UI, you may use the `simplePaginate` method to perform a single, efficient query:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate [Eloquent](#) queries. In this example, we will paginate the `App\Models\User` model and indicate that we plan to display 15 records per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
use App\Models\User;

$users = User::paginate(15);
```

Of course, you may call the `paginate` method after setting other constraints on the query, such as `where` clauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the `simplePaginate` method when paginating Eloquent models:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Similarly, you may use the `cursorPaginate` method to cursor paginate Eloquent models:

```
$users = User::where('votes', '>', 100)->cursorPaginate(15);
```

Multiple Paginator Instances per Page

Sometimes you may need to render two separate paginators on a single screen that is rendered by your application. However, if both paginator instances use the `page` query string parameter to store the current page, the two paginator's will conflict. To resolve this conflict, you may pass the name of the query string parameter you wish to use to store the paginator's current page via the third argument provided to the `paginate`, `simplePaginate`, and `cursorPaginate` methods:

```
use App\Models\User;

$users = User::where('votes', '>', 100)->paginate(
    $perPage = 15, $columns = ['*'], $pageName = 'users'
);
```

Cursor Pagination

While `paginate` and `simplePaginate` create queries using the SQL "offset" clause, cursor pagination works by constructing "where" clauses that compare the values of the ordered columns contained in the query, providing the most efficient database performance available amongst all of Laravel's pagination methods. This method of pagination is particularly well-suited for large data-sets and "infinite" scrolling user interfaces.

Unlike offset based pagination, which includes a page number in the query string of the URLs generated by the paginator, cursor based pagination places a "cursor" string in the query string. The cursor is an encoded string containing the location that the next paginated query should start paginating and the direction that it should paginate:

```
http://localhost/users?cursor=eyJpZCI6MTUsIl9wb2ludHNUb05leHRJdGVtcyI6dHJ1ZX0
```

You may create a cursor based paginator instance via the `cursorPaginate` method offered by the query builder. This method returns an instance of `Illuminate\Pagination\CursorPaginator`:

```
$users = DB::table('users')->orderBy('id')->cursorPaginate(15);
```

Once you have retrieved a cursor paginator instance, you may [display the pagination results](#) as you typically would when using the `paginate` and `simplePaginate` methods. For more information on the instance methods offered by the cursor paginator, please consult the [cursor paginator instance method documentation](#).



Your query must contain an "order by" clause in order to take advantage of cursor pagination. In addition, the columns that the query are ordered by must belong to the table you are paginating.

Cursor vs. Offset Pagination

To illustrate the differences between offset pagination and cursor pagination, let's examine some example SQL queries. Both of the following queries will both display the "second page" of results for a `users` table ordered by `id`:

```
# Offset Pagination...
select * from users order by id asc limit 15 offset 15;

# Cursor Pagination...
select * from users where id > 15 order by id asc limit 15;
```

The cursor pagination query offers the following advantages over offset pagination:

- For large data-sets, cursor pagination will offer better performance if the "order by" columns are indexed. This is because the "offset" clause scans through all previously matched data.
- For data-sets with frequent writes, offset pagination may skip records or show duplicates if results have been recently added to or deleted from the page a user is currently viewing.

However, cursor pagination has the following limitations:

- Like `simplePaginate`, cursor pagination can only be used to display "Next" and "Previous" links and does not support generating links with page numbers.
- It requires that the ordering is based on at least one unique column or a combination of columns that are unique. Columns with `null` values are not supported.
- Query expressions in "order by" clauses are supported only if they are aliased and added to the "select" clause as well.
- Query expressions with parameters are not supported.

Manually Creating a Paginator

Sometimes you may wish to create a pagination instance manually, passing it an array of items that you already have in memory. You may do so by creating either an `Illuminate\Pagination\Paginator`, `Illuminate\Pagination\LengthAwarePaginator` or `Illuminate\Pagination\CursorPaginator` instance, depending on your needs.

The `Paginator` and `CursorPaginator` classes do not need to know the total number of items in the result set; however, because of this, these classes do not have methods for retrieving the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`; however, it requires a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simplePaginate` method on the query builder, the `CursorPaginator` corresponds to the `cursorPaginate` method, and the `LengthAwarePaginator` corresponds to the `paginate` method.



When manually creating a paginator instance, you should manually "slice" the array of results you pass to the paginator. If you're unsure how to do this, check out the [array_slice](#) PHP function.

Customizing Pagination URLs

By default, links generated by the paginator will match the current request's URI. However, the paginator's `withPath` method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like `http://example.com/admin/users?page=N`, you should pass `/admin/users` to the `withPath` method:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->withPath('/admin/users');

    // ...
});
```

Appending Query String Values

You may append to the query string of pagination links using the `appends` method. For example, to append `sort=votes` to each pagination link, you should make the following call to `appends`:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->appends(['sort' => 'votes']);

    // ...
});
```

You may use the `withQueryString` method if you would like to append all of the current request's query string values to the pagination links:

```
$users = User::paginate(15)->withQueryString();
```

Appending Hash Fragments

If you need to append a "hash fragment" to URLs generated by the paginator, you may use the `fragment` method. For example, to append `#users` to the end of each pagination link, you should invoke the `fragment` method like so:

```
$users = User::paginate(15)->fragment('users');
```

Displaying Pagination Results

When calling the `paginate` method, you will receive an instance of `Illuminate\Pagination\LengthAwarePaginator`, while calling the `simplePaginate` method returns an instance of `Illuminate\Pagination\Paginator`. And, finally, calling the `cursorPaginate` method returns an instance of `Illuminate\Pagination\CursorPaginator`.

These objects provide several methods that describe the result set. In addition to these helper methods, the paginator instances are iterators and may be looped as an array. So, once you have retrieved the results, you may display the results and render the page links using [Blade](#):

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

The `links` method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper `page` query string variable. Remember, the HTML generated by the `links` method is compatible with the [Tailwind CSS framework](#).

Adjusting the Pagination Link Window

When the paginator displays pagination links, the current page number is displayed as well as links for the three pages before and after the current page. Using the `onEachSide` method, you may control how many additional links are displayed on each side of the current page within the middle, sliding window of links generated by the paginator:

```
{{ $users->onEachSide(5)->links() }}
```

Converting Results to JSON

The Laravel paginator classes implement the `Illuminate\Contracts\Support\Jsonable` interface contract and expose the `toJson` method, so it's very easy to convert your pagination results to JSON. You may also convert a paginator instance to JSON by returning it from a route or controller action:

```
use App\Models\User;

Route::get('/users', function () {
    return User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The result records are available via the `data` key in the JSON array. Here is an example of the JSON created by returning a paginator instance from a route:

```
{
    "total": 50,
    "per_page": 15,
    "current_page": 1,
    "last_page": 4,
    "first_page_url": "http://laravel.app?page=1",
    "last_page_url": "http://laravel.app?page=4",
    "next_page_url": "http://laravel.app?page=2",
    "prev_page_url": null,
    "path": "http://laravel.app",
    "from": 1,
    "to": 15,
    "data": [
        {
            // Record...
        },
        {
            // Record...
        }
    ]
}
```

Customizing the Pagination View

By default, the views rendered to display the pagination links are compatible with the [Tailwind CSS](#) framework. However, if you are not using Tailwind, you are free to define your own views to render these links. When calling the `links` method on a paginator instance, you may pass the view name as the first argument to the method:

```
{{ $paginator->links('view.name') }}

<!-- Passing additional data to the view... -->
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

However, the easiest way to customize the pagination views is by exporting them to your `resources/views/vendor` directory using the `vendor:publish` command:

```
php artisan vendor:publish --tag=laravel-pagination
```

This command will place the views in your application's `resources/views/vendor/pagination` directory. The `tailwind.blade.php` file within this directory corresponds to the default pagination view. You may edit this file to modify the pagination HTML.

If you would like to designate a different file as the default pagination view, you may invoke the paginator's `defaultView` and `defaultSimpleView` methods within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Paginator::defaultView('view-name');

        Paginator::defaultSimpleView('view-name');
    }
}
```

Using Bootstrap

Laravel includes pagination views built using [Bootstrap CSS](#). To use these views instead of the default Tailwind views, you may call the paginator's `useBootstrapFour` or `useBootstrapFive` methods within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
use Illuminate\Pagination\Paginator;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Paginator::useBootstrapFive();
    Paginator::useBootstrapFour();
}
```

Paginator / LengthAwarePaginator Instance Methods

Each paginator instance provides additional pagination information via the following methods:

Method	Description
<code>\$paginator->count()</code>	Get the number of items for the current page.
<code>\$paginator->currentPage()</code>	Get the current page number.
<code>\$paginator->firstItem()</code>	Get the result number of the first item in the results.
<code>\$paginator->getOptions()</code>	Get the paginator options.
<code>\$paginator->getUrlRange(\$start, \$end)</code>	Create a range of pagination URLs.
<code>\$paginator->hasPages()</code>	Determine if there are enough items to split into multiple pages.
<code>\$paginator->hasMorePages()</code>	Determine if there are more items in the data store.
<code>\$paginator->items()</code>	Get the items for the current page.
<code>\$paginator->lastItem()</code>	Get the result number of the last item in the results.
<code>\$paginator->lastPage()</code>	Get the page number of the last available page. (Not available when using <code>simplePaginate</code>).
<code>\$paginator->nextPageUrl()</code>	Get the URL for the next page.
<code>\$paginator->onFirstPage()</code>	Determine if the paginator is on the first page.
<code>\$paginator->perPage()</code>	The number of items to be shown per page.
<code>\$paginator->previousPageUrl()</code>	Get the URL for the previous page.
<code>\$paginator->total()</code>	Determine the total number of matching items in the data store. (Not available when using <code>simplePaginate</code>).
<code>\$paginator->url(\$page)</code>	Get the URL for a given page number.
<code>\$paginator->getPageName()</code>	Get the query string variable used to store the page.
<code>\$paginator->setPageName(\$name)</code>	Set the query string variable used to store the page.
<code>\$paginator->through(\$callback)</code>	Transform each item using a callback.

Cursor Paginator Instance Methods

Each cursor paginator instance provides additional pagination information via the following methods:

Method	Description
\$paginator->count()	Get the number of items for the current page.
\$paginator->cursor()	Get the current cursor instance.
\$paginator->getOptions()	Get the paginator options.
\$paginator->hasPages()	Determine if there are enough items to split into multiple pages.
\$paginator->hasMorePages()	Determine if there are more items in the data store.
\$paginator->getCursorName()	Get the query string variable used to store the cursor.
\$paginator->items()	Get the items for the current page.
\$paginator->nextCursor()	Get the cursor instance for the next set of items.
\$paginator->nextPageUrl()	Get the URL for the next page.
\$paginator->onFirstPage()	Determine if the paginator is on the first page.
\$paginator->onLastPage()	Determine if the paginator is on the last page.
\$paginator->perPage()	The number of items to be shown per page.
\$paginator->previousCursor()	Get the cursor instance for the previous set of items.
\$paginator->previousPageUrl()	Get the URL for the previous page.
\$paginator->setCursorName()	Set the query string variable used to store the cursor.
\$paginator->url(\$cursor)	Get the URL for a given cursor instance.

Database: Migrations

- [Introduction](#)
- [Generating Migrations](#)
 - [Squashing Migrations](#)
- [Migration Structure](#)
- [Running Migrations](#)
 - [Rolling Back Migrations](#)
- [Tables](#)
 - [Creating Tables](#)
 - [Updating Tables](#)
 - [Renaming / Dropping Tables](#)
- [Columns](#)
 - [Creating Columns](#)
 - [Available Column Types](#)
 - [Column Modifiers](#)
 - [Modifying Columns](#)
 - [Renaming Columns](#)
 - [Dropping Columns](#)
- [Indexes](#)
 - [Creating Indexes](#)
 - [Renaming Indexes](#)
 - [Dropping Indexes](#)
 - [Foreign Key Constraints](#)
- [Events](#)

Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

Generating Migrations

You may use the `make:migration` Artisan command to generate a database migration. The new migration will be placed in your `database/migrations` directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

Laravel will use the name of the migration to attempt to guess the name of the table and whether or not the migration will be creating a new table. If Laravel is able to determine the table name from the migration name, Laravel will pre-fill the generated migration file with the specified table. Otherwise, you may simply specify the table in the migration file manually.

If you would like to specify a custom path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The given path should be relative to your application's base path.



Migration stubs may be customized using [stub publishing](#).

Squashing Migrations

As you build your application, you may accumulate more and more migrations over time. This can lead to your `database/migrations` directory becoming bloated with potentially hundreds of migrations. If you would like, you may "squash" your migrations into a single SQL file. To get started, execute the `schema:dump` command:

```
php artisan schema:dump

# Dump the current database schema and prune all existing migrations...
php artisan schema:dump --prune
```

When you execute this command, Laravel will write a "schema" file to your application's `database/schema` directory. The schema file's name will correspond to the database connection. Now, when you attempt to migrate your database and no other migrations have been executed, Laravel will first execute the SQL statements in the schema file of the database connection you are using. After executing the schema file's SQL statements, Laravel will execute any remaining migrations that were not part of the schema dump.

If your application's tests use a different database connection than the one you typically use during local development, you should ensure you have dumped a schema file using that database connection so that your tests are able to build your database. You may wish to do this after dumping the database connection you typically use during local development:

```
php artisan schema:dump
php artisan schema:dump --database=testing --prune
```

You should commit your database schema file to source control so that other new developers on your team may quickly create your application's initial database structure.



Migration squashing is only available for the MySQL, PostgreSQL, and SQLite databases and utilizes the database's command-line client.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should reverse the operations performed by the `up` method.

Within both of these methods, you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, [check out its documentation](#). For example, the following migration creates a `flights` table:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::drop('flights');
    }
};
```

Setting the Migration Connection

If your migration will be interacting with a database connection other than your application's default database connection, you should set the `$connection` property of your migration:

```
/**
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Run the migrations.
 */
public function up(): void
{
    // ...
}
```

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

If you would like to see which migrations have run thus far, you may use the `migrate:status` Artisan command:

```
php artisan migrate:status
```

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the `--pretend` flag to the `migrate` command:

```
php artisan migrate --pretend
```

Isolating Migration Execution

If you are deploying your application across multiple servers and running migrations as part of your deployment process, you likely do not want two servers attempting to migrate the database at the same time. To avoid this, you may use the `isolated` option when invoking the `migrate` command.

When the `isolated` option is provided, Laravel will acquire an atomic lock using your application's cache driver before attempting to run your migrations. All other attempts to run the `migrate` command while that lock is held will not execute; however, the command will still exit with a successful exit status code:

```
php artisan migrate --isolated
```



To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`, or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Forcing Migrations to Run in Production

Some migration operations are destructive, which means they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before the commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To roll back the latest migration operation, you may use the `rollback` Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may roll back a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will roll back the last five migrations:

```
php artisan migrate:rollback --step=5
```

You may roll back a specific "batch" of migrations by providing the `batch` option to the `rollback` command, where the `batch` option corresponds to a batch value within your application's `migrations` database table. For example, the

following command will roll back all migrations in batch three:

```
php artisan migrate:rollback --batch=3
```

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the `--pretend` flag to the `migrate:rollback` command:

```
php artisan migrate:rollback --pretend
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Roll Back and Migrate Using a Single Command

The `migrate:refresh` command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

# Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

You may roll back and re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will roll back and re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Tables and Migrate

The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

By default, the `migrate:fresh` command only drops tables from the default database connection. However, you may use the `--database` option to specify the database connection that should be migrated. The database connection name should correspond to a connection defined in your application's [database configuration file](#):

```
php artisan migrate:fresh --database=admin
```



The `migrate:fresh` command will drop all database tables regardless of their prefix. This command should be used with caution when developing on a database that is shared with other applications.

Tables

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments: the first is the name of the table, while the second is a closure which receives a `Blueprint` object that may be used to define the new table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

When creating the table, you may use any of the schema builder's [column methods](#) to define the table's columns.

Determining Table / Column Existence

You may determine the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

Database Connection and Table Options

If you want to perform a schema operation on a database connection that is not your application's default connection, use the `connection` method:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

In addition, a few other properties and methods may be used to define other aspects of the table's creation. The `engine` property may be used to specify the table's storage engine when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    // ...
});
```

The `charset` and `collation` properties may be used to specify the character set and collation for the created table when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

The `temporary` method may be used to indicate that the table should be "temporary". Temporary tables are only visible to the current connection's database session and are dropped automatically when the connection is closed:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

If you would like to add a "comment" to a database table, you may invoke the `comment` method on the table instance. Table comments are currently only supported by MySQL and Postgres:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->comment('Business calculations');

    // ...
});
```

Updating Tables

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives a `Blueprint` instance you may use to add columns or indexes to the table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Renaming Tables With Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives an `Illuminate\Database\Schema\Blueprint` instance you may use to add columns to the table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Available Column Types

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

<code>bigIncrements</code>	<code>jsonb</code>	<code>string</code>
<code>bigInteger</code>	<code>lineString</code>	<code>text</code>
<code>binary</code>	<code>longText</code>	<code>timeTz</code>
<code>boolean</code>	<code>macAddress</code>	<code>time</code>
<code>char</code>	<code>mediumIncrements</code>	<code>timestampTz</code>
<code>dateTimeTz</code>	<code>mediumInteger</code>	<code>timestamp</code>
<code>dateTime</code>	<code>mediumText</code>	<code>timestampsTz</code>
<code>date</code>	<code>morphs</code>	<code>timestamps</code>
<code>decimal</code>	<code>multiLineString</code>	<code>tinyIncrements</code>
<code>double</code>	<code>multiPoint</code>	<code>tinyInteger</code>
<code>enum</code>	<code>multiPolygon</code>	<code>tinyText</code>
<code>float</code>	<code>nullableMorphs</code>	<code>unsignedBigInteger</code>
<code>foreignId</code>	<code>nullableTimestamps</code>	<code>unsignedDecimal</code>
<code>foreignIdFor</code>	<code>nullableUlidMorphs</code>	<code>unsignedInteger</code>
<code>foreignUlid</code>	<code>nullableUuidMorphs</code>	<code>unsignedMediumInteger</code>
<code>foreignUuid</code>	<code>point</code>	<code>unsignedSmallInteger</code>
<code>geometryCollection</code>	<code>polygon</code>	<code>unsignedTinyInteger</code>
<code>geometry</code>	<code>rememberToken</code>	<code>ulidMorphs</code>
<code>id</code>	<code>set</code>	<code>uuidMorphs</code>
<code>increments</code>	<code>smallIncrements</code>	<code>ulid</code>
<code>integer</code>	<code>smallInteger</code>	<code>uuid</code>
<code>ipAddress</code>	<code>softDeletesTz</code>	<code>year</code>
<code>json</code>	<code>softDeletes</code>	

`bigIncrements()`

The `bigIncrements` method creates an auto-incrementing `UNSIGNED BIGINT` (primary key) equivalent column:

```
$table->bigIncrements('id');
```

bigInteger()

The `bigInteger` method creates a `BIGINT` equivalent column:

```
$table->bigInteger('votes');
```

binary()

The `binary` method creates a `BLOB` equivalent column:

```
$table->binary('photo');
```

boolean()

The `boolean` method creates a `BOOLEAN` equivalent column:

```
$table->boolean('confirmed');
```

char()

The `char` method creates a `CHAR` equivalent column with of a given length:

```
$table->char('name', 100);
```

dateTimeTz()

The `dateTimeTz` method creates a `DATETIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->dateTimeTz('created_at', $precision = 0);
```

dateTime()

The `dateTime` method creates a `DATETIME` equivalent column with an optional precision (total digits):

```
$table->dateTime('created_at', $precision = 0);
```

date()

The `date` method creates a `DATE` equivalent column:

```
$table->date('created_at');
```

decimal()

The `decimal` method creates a `DECIMAL` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

double()

The `double` method creates a `DOUBLE` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->double('amount', 8, 2);
```

enum()

The `enum` method creates a `ENUM` equivalent column with the given valid values:

```
$table->enum('difficulty', ['easy', 'hard']);
```

float()

The `float` method creates a `FLOAT` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->float('amount', 8, 2);
```

foreignId()

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->foreignId('user_id');
```

foreignIdFor()

The `foreignIdFor` method adds a `{column}_id` equivalent column for a given model class. The column type will be `UNSIGNED BIGINT`, `CHAR(36)`, or `CHAR(26)` depending on the model key type:

```
$table->foreignIdFor(User::class);
```

foreignUlid()

The `foreignUlid` method creates a `ULID` equivalent column:

```
$table->foreignUlid('user_id');
```

foreignUuid()

The `foreignUuid` method creates a `UUID` equivalent column:

```
$table->foreignUuid('user_id');
```

geometryCollection()

The `geometryCollection` method creates a `GEOMETRYCOLLECTION` equivalent column:

```
$table->geometryCollection('positions');
```

geometry()

The `geometry` method creates a `GEOMETRY` equivalent column:

```
$table->geometry('positions');
```

id()

The `id` method is an alias of the `bigIncrements` method. By default, the method will create an `id` column; however, you may pass a column name if you would like to assign a different name to the column:

```
$table->id();
```

`increments()`

The `increments` method creates an auto-incrementing `UNSIGNED INTEGER` equivalent column as a primary key:

```
$table->increments('id');
```

`integer()`

The `integer` method creates an `INTEGER` equivalent column:

```
$table->integer('votes');
```

`ipAddress()`

The `ipAddress` method creates a `VARCHAR` equivalent column:

```
$table->ipAddress('visitor');
```

When using Postgres, an `INET` column will be created.

`json()`

The `json` method creates a `JSON` equivalent column:

```
$table->json('options');
```

`jsonb()`

The `jsonb` method creates a `JSONB` equivalent column:

```
$table->jsonb('options');
```

`lineString()`

The `lineString` method creates a `LINESTRING` equivalent column:

```
$table->lineString('positions');
```

`longText()`

The `longText` method creates a `LONGTEXT` equivalent column:

```
$table->longText('description');
```

`macAddress()`

The `macAddress` method creates a column that is intended to hold a MAC address. Some database systems, such as PostgreSQL, have a dedicated column type for this type of data. Other database systems will use a string equivalent column:

```
$table->macAddress('device');
```

mediumIncrements()

The `mediumIncrements` method creates an auto-incrementing `UNSIGNED MEDIUMINT` equivalent column as a primary key:

```
$table->mediumIncrements('id');
```

mediumInteger()

The `mediumInteger` method creates a `MEDIUMINT` equivalent column:

```
$table->mediumInteger('votes');
```

mediumText()

The `mediumText` method creates a `MEDIUMTEXT` equivalent column:

```
$table->mediumText('description');
```

morphs()

The `morphs` method is a convenience method that adds a `{column}_id` equivalent column and a `{column}_type` `VARCHAR` equivalent column. The column type for the `{column}_id` will be `UNSIGNED BIGINT`, `CHAR(36)`, or `CHAR(26)` depending on the model key type.

This method is intended to be used when defining the columns necessary for a polymorphic [Eloquent relationship](#). In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->morphs('taggable');
```

multiLineString()

The `multiLineString` method creates a `MULTILINESTRING` equivalent column:

```
$table->multiLineString('positions');
```

multiPoint()

The `multiPoint` method creates a `MULTIPOINT` equivalent column:

```
$table->multiPoint('positions');
```

multiPolygon()

The `multiPolygon` method creates a `MULTIPOLYGON` equivalent column:

```
$table->multiPolygon('positions');
```

nullableTimestamps()

The `nullableTimestamps` method is an alias of the [timestamps](#) method:

```
$table->nullableTimestamps(0);
```

nullableMorphs()

The method is similar to the [morphs](#) method; however, the columns that are created will be "nullable":

```
$table->nullableMorphs('taggable');
```

nullableUlidMorphs()

The method is similar to the [ulidMorphs](#) method; however, the columns that are created will be "nullable":

```
$table->nullableUlidMorphs('taggable');
```

nullableUuidMorphs()

The method is similar to the [uuidMorphs](#) method; however, the columns that are created will be "nullable":

```
$table->nullableUuidMorphs('taggable');
```

point()

The `point` method creates a `POINT` equivalent column:

```
$table->point('position');
```

polygon()

The `polygon` method creates a `POLYGON` equivalent column:

```
$table->polygon('position');
```

rememberToken()

The `rememberToken` method creates a nullable, `VARCHAR(100)` equivalent column that is intended to store the current "remember me" [authentication token](#):

```
$table->rememberToken();
```

set()

The `set` method creates a `SET` equivalent column with the given list of valid values:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

The `smallIncrements` method creates an auto-incrementing `UNSIGNED SMALLINT` equivalent column as a primary key:

```
$table->smallIncrements('id');
```

smallInteger()

The `smallInteger` method creates a `SMALLINT` equivalent column:

```
$table->smallInteger('votes');
```

`softDeletesTz()`

The `softDeletesTz` method adds a nullable `deleted_at` `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

`softDeletes()`

The `softDeletes` method adds a nullable `deleted_at` `TIMESTAMP` equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

`string()`

The `string` method creates a `VARCHAR` equivalent column of the given length:

```
$table->string('name', 100);
```

`text()`

The `text` method creates a `TEXT` equivalent column:

```
$table->text('description');
```

`timeTz()`

The `timeTz` method creates a `TIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timeTz('sunrise', $precision = 0);
```

`time()`

The `time` method creates a `TIME` equivalent column with an optional precision (total digits):

```
$table->time('sunrise', $precision = 0);
```

`timestampTz()`

The `timestampTz` method creates a `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timestampTz('added_at', $precision = 0);
```

`timestamp()`

The `timestamp` method creates a `TIMESTAMP` equivalent column with an optional precision (total digits):

```
$table->timestamp('added_at', $precision = 0);
```

timestampsTz()

The `timestampsTz` method creates `created_at` and `updated_at` `TIMESTAMP` (with timezone) equivalent columns with an optional precision (total digits):

```
$table->timestampsTz($precision = 0);
```

timestamps()

The `timestamps` method creates `created_at` and `updated_at` `TIMESTAMP` equivalent columns with an optional precision (total digits):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

The `tinyIncrements` method creates an auto-incrementing `UNSIGNED TINYINT` equivalent column as a primary key:

```
$table->tinyIncrements('id');
```

tinyInteger()

The `tinyInteger` method creates a `TINYINT` equivalent column:

```
$table->tinyInteger('votes');
```

tinyText()

The `tinyText` method creates a `TINYTEXT` equivalent column:

```
$table->tinyText('notes');
```

unsignedBigInteger()

The `unsignedBigInteger` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

The `unsignedDecimal` method creates an `UNSIGNED DECIMAL` equivalent column with an optional precision (total digits) and scale (decimal digits):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

unsignedInteger()

The `unsignedInteger` method creates an `UNSIGNED INTEGER` equivalent column:

```
$table->unsignedInteger('votes');
```

unsignedMediumInteger()

The `unsignedMediumInteger` method creates an `UNSIGNED MEDIUMINT` equivalent column:

```
$table->unsignedMediumInteger('votes');
```

unsignedSmallInteger()

The `unsignedSmallInteger` method creates an `UNSIGNED SMALLINT` equivalent column:

```
$table->unsignedSmallInteger('votes');
```

unsignedTinyInteger()

The `unsignedTinyInteger` method creates an `UNSIGNED TINYINT` equivalent column:

```
$table->unsignedTinyInteger('votes');
```

ulidMorphs()

The `ulidMorphs` method is a convenience method that adds a `{column}_id` `CHAR(26)` equivalent column and a `{column}_type` `VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic [Eloquent relationship](#) that use ULID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->ulidMorphs('taggable');
```

uuidMorphs()

The `uuidMorphs` method is a convenience method that adds a `{column}_id` `CHAR(36)` equivalent column and a `{column}_type` `VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic [Eloquent relationship](#) that use UUID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->uuidMorphs('taggable');
```

ulid()

The `ulid` method creates a `ULID` equivalent column:

```
$table->ulid('id');
```

uuid()

The `uuid` method creates a `UUID` equivalent column:

```
$table->uuid('id');
```

year()

The `year` method creates a `YEAR` equivalent column:

```
$table->year('birth_year');
```

Column Modifiers

In addition to the column types listed above, there are several column "modifiers" you may use when adding a column to a database table. For example, to make the column "nullable", you may use the `nullable` method:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

The following table contains all of the available column modifiers. This list does not include [index modifiers](#):

Modifier	Description
<code>->after('column')</code>	Place the column "after" another column (MySQL).
<code>->autoIncrement()</code>	Set INTEGER columns as auto-incrementing (primary key).
<code>->charset('utf8mb4')</code>	Specify a character set for the column (MySQL).
<code>->collation('utf8mb4_unicode_ci')</code>	Specify a collation for the column (MySQL/PostgreSQL/SQL Server).
<code>->comment('my comment')</code>	Add a comment to a column (MySQL/PostgreSQL).
<code>->default(\$value)</code>	Specify a "default" value for the column.
<code>->first()</code>	Place the column "first" in the table (MySQL).
<code>->from(\$integer)</code>	Set the starting value of an auto-incrementing field (MySQL / PostgreSQL).
<code>->invisible()</code>	Make the column "invisible" to SELECT * queries (MySQL).
<code>->nullable(\$value = true)</code>	Allow NULL values to be inserted into the column.
<code>->storedAs(\$expression)</code>	Create a stored generated column (MySQL / PostgreSQL).
<code>->unsigned()</code>	Set INTEGER columns as UNSIGNED (MySQL).
<code>->useCurrent()</code>	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value.
<code>->useCurrentOnUpdate()</code>	Set TIMESTAMP columns to use CURRENT_TIMESTAMP when a record is updated (MySQL).
<code>->virtualAs(\$expression)</code>	Create a virtual generated column (MySQL / PostgreSQL / SQLite).
<code>->generatedAs(\$expression)</code>	Create an identity column with specified sequence options (PostgreSQL).
<code>->always()</code>	Defines the precedence of sequence values over input for an identity column (PostgreSQL).
<code>->isGeometry()</code>	Set spatial column type to geometry - the default type is geography (PostgreSQL).

Default Expressions

The `default` modifier accepts a value or an `Illuminate\Database\Query\Expression` instance. Using an `Expression` instance will prevent Laravel from wrapping the value in quotes and allow you to use database specific functions. One situation where this is particularly useful is when you need to assign default values to JSON columns:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('JSON_ARRAY()'));
            $table->timestamps();
        });
    }
};
```



Support for default expressions depends on your database driver, database version, and the field type. Please refer to your database's documentation.

Column Order

When using the MySQL database, the `after` method may be used to add columns after an existing column in the schema:

```
$table->after('password', function (Blueprint $table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Modifying Columns

The `change` method allows you to modify the type and attributes of existing columns. For example, you may wish to increase the size of a `string` column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50. To accomplish this, we simply define the new state of the column and then call the `change` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

When modifying a column, you must explicitly include all of the modifiers you want to keep on the column definition - any missing attribute will be dropped. For example, to retain the `unsigned`, `default`, and `comment` attributes, you must call each modifier explicitly when changing the column:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')->unsigned()->default(1)->comment('my comment')->change();
});
```

Modifying Columns on SQLite

If your application is utilizing an SQLite database, you must install the `doctrine/dbal` package using the Composer package manager before modifying a column. The Doctrine DBAL library is used to determine the current state of the column and to create the SQL queries needed to make the requested changes to your column:

```
composer require doctrine/dbal
```

If you plan to modify columns created using the `timestamp` method, you must also add the following configuration to your application's `config/database.php` configuration file:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

When using the `doctrine/dbal` package, the following column types can be modified:

`bigInteger`, `binary`, `boolean`, `char`, `date`, `dateTime`, `dateTimeTz`, `decimal`,
`double`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`,
`time`, `tinyText`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger`,
`ulid`, and `uuid`.

Renaming Columns

To rename a column, you may use the `renameColumn` method provided by the schema builder:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

Renaming Columns on Legacy Databases

If you are running a database installation older than one of the following releases, you should ensure that you have installed the `doctrine/dbal` library via the Composer package manager before renaming a column:

- MySQL < `8.0.3`
- MariaDB < `10.5.2`
- SQLite < `3.25.0`

Dropping Columns

To drop a column, you may use the `dropColumn` method on the schema builder:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

You may drop multiple columns from a table by passing an array of column names to the `dropColumn` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

Dropping Columns on Legacy Databases

If you are running a version of SQLite prior to 3.35.0, you must install the `doctrine/dbal` package via the Composer package manager before the `dropColumn` method may be used. Dropping or modifying multiple columns within a single migration while using this package is not supported.

Available Command Aliases

Laravel provides several convenient methods related to dropping common types of columns. Each of these methods is described in the table below:

Command	Description
<code>\$table->dropMorphs('morphable');</code>	Drop the <code>morphable_id</code> and <code>morphable_type</code> columns.
<code>\$table->dropRememberToken();</code>	Drop the <code>remember_token</code> column.
<code>\$table->dropSoftDeletes();</code>	Drop the <code>deleted_at</code> column.
<code>\$table->dropSoftDeletesTz();</code>	Alias of <code>dropSoftDeletes()</code> method.
<code>\$table->dropTimestamps();</code>	Drop the <code>created_at</code> and <code>updated_at</code> columns.
<code>\$table->dropTimestampsTz();</code>	Alias of <code>dropTimestamps()</code> method.

Indexes

Creating Indexes

The Laravel schema builder supports several types of indexes. The following example creates a new `email` column and specifies that its values should be unique. To create the index, we can chain the `unique` method onto the column definition:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
});
```

Alternatively, you may create the index after defining the column. To do so, you should call the `unique` method on the schema builder blueprint. This method accepts the name of the column that should receive a unique index:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound (or composite) index:

```
$table->index(['account_id', 'created_at']);
```

When creating an index, Laravel will automatically generate an index name based on the table, column names, and the index type, but you may pass a second argument to the method to specify the index name yourself:

```
$table->unique('email', 'unique_email');
```

Available Index Types

Laravel's schema builder blueprint class provides methods for creating each type of index supported by Laravel. Each index method accepts an optional second argument to specify the name of the index. If omitted, the name will be derived from the names of the table and column(s) used for the index, as well as the index type. Each of the available index methods is described in the table below:

Command	Description
<code>\$table->primary('id');</code>	Adds a primary key.
<code>\$table->primary(['id', 'parent_id']);</code>	Adds composite keys.
<code>\$table->unique('email');</code>	Adds a unique index.
<code>\$table->index('state');</code>	Adds an index.
<code>\$table->fullText('body');</code>	Adds a full text index (MySQL/PostgreSQL).
<code>\$table->fullText('body')->language('english');</code>	Adds a full text index of the specified language (PostgreSQL).
<code>\$table->spatialIndex('location');</code>	Adds a spatial index (except SQLite).

Index Lengths and MySQL / MariaDB

By default, Laravel uses the `utf8mb4` character set. If you are running a version of MySQL older than the 5.7.7 release or MariaDB older than the 10.2.2 release, you may need to manually configure the default string length generated by migrations in order for MySQL to create indexes for them. You may configure the default string length by calling the `Schema::defaultStringLength` method within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Schema::defaultStringLength(191);
}
```

Alternatively, you may enable the `innodb_large_prefix` option for your database. Refer to your database's documentation for instructions on how to properly enable this option.

Renaming Indexes

To rename an index, you may use the `renameIndex` method provided by the schema builder blueprint. This method accepts the current index name as its first argument and the desired name as its second argument:

```
$table->renameIndex('from', 'to')
```



If your application is utilizing an SQLite database, you must install the `doctrine/dbal` package via the Composer package manager before the `renameIndex` method may be used.

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns an index name based on the table name, the name of the indexed column, and the index type. Here are some examples:

Command	Description
<code>\$table->dropPrimary('users_id_primary');</code>	Drop a primary key from the "users" table.
<code>\$table->dropUnique('users_email_unique');</code>	Drop a unique index from the "users" table.
<code>\$table->dropIndex('geo_state_index');</code>	Drop a basic index from the "geo" table.
<code>\$table->dropFullText('posts_body_fulltext');</code>	Drop a full text index from the "posts" table.
<code>\$table->dropSpatialIndex('geo_location_spatialindex');</code>	Drop a spatial index from the "geo" table (except SQLite).

If you pass an array of columns into a method that drops indexes, the conventional index name will be generated based on the table name, columns, and index type:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

Since this syntax is rather verbose, Laravel provides additional, terser methods that use conventions to provide a better developer experience. When using the `foreignId` method to create your column, the example above can be rewritten like so:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column, while the `constrained` method will use conventions to determine the table and column being referenced. If your table name does not match Laravel's conventions, you may manually provide it to the `constrained` method. In addition, the name that should be assigned to the generated index may be specified as well:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained(
        table: 'users', indexName: 'posts_user_id'
    );
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreignId('user_id')
    ->constrained()
    ->onUpdate('cascade')
    ->onDelete('cascade');
```

An alternative, expressive syntax is also provided for these actions:

Method	Description
<code>\$table->cascadeOnUpdate();</code>	Updates should cascade.
<code>\$table->restrictOnUpdate();</code>	Updates should be restricted.
<code>\$table->noActionOnUpdate();</code>	No action on updates.
<code>\$table->cascadeonDelete();</code>	Deletes should cascade.
<code>\$table->restrictonDelete();</code>	Deletes should be restricted.
<code>\$table->>nullonDelete();</code>	Deletes should set the foreign key value to null.

Any additional column modifiers must be called before the `constrained` method:

```
$table->foreignId('user_id')
    ->nullable()
    ->constrained();
```

Dropping Foreign Keys

To drop a foreign key, you may use the `dropForeign` method, passing the name of the foreign key constraint to be deleted as an argument. Foreign key constraints use the same naming convention as indexes. In other words, the foreign key constraint name is based on the name of the table and the columns in the constraint, followed by a `_foreign` suffix:

```
$table->dropForeign('posts_user_id_foreign');
```

Alternatively, you may pass an array containing the column name that holds the foreign key to the `dropForeign` method. The array will be converted to a foreign key constraint name using Laravel's constraint naming conventions:

```
$table->dropForeign(['user_id']);
```

Toggling Foreign Key Constraints

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();

Schema::withoutForeignKeyConstraints(function () {
    // Constraints disabled within this closure...
});
```



SQLite disables foreign key constraints by default. When using SQLite, make sure to [enable foreign key support](#) in your database configuration before attempting to create them in your migrations. In addition, SQLite only supports foreign keys upon creation of the table and [not when tables are altered](#).

Events

For convenience, each migration operation will dispatch an [event](#). All of the following events extend the base `Illuminate\Database\Events\MigrationEvent` class:

Class	Description
<code>Illuminate\Database\Events\MigrationsStarted</code>	A batch of migrations is about to be executed.
<code>Illuminate\Database\Events\MigrationsEnded</code>	A batch of migrations has finished executing.
<code>Illuminate\Database\Events\MigrationStarted</code>	A single migration is about to be executed.
<code>Illuminate\Database\Events\MigrationEnded</code>	A single migration has finished executing.
<code>Illuminate\Database\Events\SchemaDumped</code>	A database schema dump has completed.
<code>Illuminate\Database\Events\SchemaLoaded</code>	An existing database schema dump has loaded.

Database: Seeding

- [Introduction](#)
- [Writing Seeders](#)
 - [Using Model Factories](#)
 - [Calling Additional Seeders](#)
 - [Muting Model Events](#)
- [Running Seeders](#)

Introduction

Laravel includes the ability to seed your database with data using seed classes. All seed classes are stored in the `database/seeders` directory. By default, a `DatabaseSeeder` class is defined for you. From this class, you may use the `call` method to run other seed classes, allowing you to control the seeding order.



Mass assignment protection is automatically disabled during database seeding.

Writing Seeders

To generate a seeder, execute the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeders` directory:

```
php artisan make:seeder UserSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the [query builder](#) to manually insert data or you may use [Eloquent model factories](#).

As an example, let's modify the default `DatabaseSeeder` class and add a database insert statement to the `run` method:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     */
    public function run(): void
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@example.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```



You may type-hint any dependencies you need within the `run` method's signature. They will automatically be resolved via the Laravel [service container](#).

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use [model factories](#) to conveniently generate large amounts of database records. First, review the [model factory documentation](#) to learn how to define your factories.

For example, let's create 50 users that each has one related post:

```
use App\Models\User;

/**
 * Run the database seeders.
 */
public function run(): void
{
    User::factory()
        ->count(50)
        ->hasPosts(1)
        ->create();
}
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you may use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes too large.

The `call` method accepts an array of seeder classes that should be executed:

```
/** 
 * Run the database seeders.
 */
public function run(): void
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

Muting Model Events

While running seeds, you may want to prevent models from dispatching events. You may achieve this using the `WithoutModelEvents` trait. When used, the `WithoutModelEvents` trait ensures no model events are dispatched, even if additional seed classes are executed via the `call` method:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;

class DatabaseSeeder extends Seeder
{
    use WithoutModelEvents;

    /**
     * Run the database seeders.
     */
    public function run(): void
    {
        $this->call([
            UserSeeder::class,
        ]);
    }
}
```

Running Seeders

You may execute the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `Database\Seeders\DatabaseSeeder` class, which may in turn invoke other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed

php artisan db:seed --class=UserSeeder
```

You may also seed your database using the `migrate:fresh` command in combination with the `--seed` option, which will drop all tables and re-run all of your migrations. This command is useful for completely re-building your database. The

--seeder option may be used to specify a specific seeder to run:

```
php artisan migrate:fresh --seed  
php artisan migrate:fresh --seed UserSeeder
```

Forcing Seeders to Run in Production

Some seeding operations may cause you to alter or lose data. In order to protect you from running seeding commands against your production database, you will be prompted for confirmation before the seeders are executed in the production environment. To force the seeders to run without a prompt, use the --force flag:

```
php artisan db:seed --force
```

Redis

- [Introduction](#)
- [Configuration](#)
 - [Clusters](#)
 - [Predis](#)
 - [PhpRedis](#)
- [Interacting With Redis](#)
 - [Transactions](#)
 - [Pipelining Commands](#)
- [Pub / Sub](#)

Introduction

[Redis](#) is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Before using Redis with Laravel, we encourage you to install and use the [PhpRedis](#) PHP extension via PECL. The extension is more complex to install compared to "user-land" PHP packages but may yield better performance for applications that make heavy use of Redis. If you are using [Laravel Sail](#), this extension is already installed in your application's Docker container.

If you are unable to install the PhpRedis extension, you may install the [predis/predis](#) package via Composer. Predis is a Redis client written entirely in PHP and does not require any additional extensions:

```
composer require predis/predis
```

Configuration

You may configure your application's Redis settings via the [config/database.php](#) configuration file. Within this file, you will see a `redis` array containing the Redis servers utilized by your application:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

    'cache' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_CACHE_DB', 1),
    ],
],
```

Each Redis server defined in your configuration file is required to have a name, host, and a port unless you define a single URL to represent the Redis connection:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'default' => [
        'url' => 'tcp://127.0.0.1:6379?database=0',
    ],
    'cache' => [
        'url' => 'tls://user:password@127.0.0.1:6380?database=1',
    ],
],
```

Configuring the Connection Scheme

By default, Redis clients will use the `tcp` scheme when connecting to your Redis servers; however, you may use TLS / SSL encryption by specifying a `scheme` configuration option in your Redis server's configuration array:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'default' => [
        'scheme' => 'tls',
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],
],
```

Clusters

If your application is utilizing a cluster of Redis servers, you should define these clusters within a `clusters` key of your Redis configuration. This configuration key does not exist by default so you will need to create it within your application's `config/database.php` configuration file:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD'),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],
],
```

By default, clusters will perform client-side sharding across your nodes, allowing you to pool nodes and create a large amount of available RAM. However, client-side sharding does not handle failover; therefore, it is primarily suited for transient cached data that is available from another primary data store.

If you would like to use native Redis clustering instead of client-side sharding, you may specify this by setting the `options.cluster` configuration value to `redis` within your application's `config/database.php` configuration file:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],
    'clusters' => [
        // ...
    ],
],
```

Predis

If you would like your application to interact with Redis via the Predis package, you should ensure the `REDIS_CLIENT` environment variable's value is `predis`:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'predis'),
    // ...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, Predis supports additional [connection parameters](#) that may be defined for each of your Redis servers. To utilize these additional configuration options, add them to your Redis server configuration in your application's `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD'),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
],
```

The Redis Facade Alias

Laravel's `config/app.php` configuration file contains an `aliases` array which defines all of the class aliases that will be registered by the framework. By default, no `Redis` alias is included because it would conflict with the `Redis` class name provided by the `PhpRedis` extension. If you are using the `Predis` client and would like to add a `Redis` alias, you may add it to the `aliases` array in your application's `config/app.php` configuration file:

```
'aliases' => Facade::defaultAliases()->merge([
    'Redis' => Illuminate\Support\Facades\Redis::class,
])->toArray(),
```

PhpRedis

By default, Laravel will use the `PhpRedis` extension to communicate with Redis. The client that Laravel will use to communicate with Redis is dictated by the value of the `redis.client` configuration option, which typically reflects the value of the `REDIS_CLIENT` environment variable:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    // Rest of Redis configuration...
],
```

In addition to the default `scheme`, `host`, `port`, `database`, and `password` server configuration options, `PhpRedis` supports the following additional connection parameters: `name`, `persistent`, `persistent_id`, `prefix`, `read_timeout`, `retry_interval`, `timeout`, and `context`. You may add any of these options to your Redis server configuration in the `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD'),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
    'context' => [
        // 'auth' => ['username', 'secret'],
        // 'stream' => ['verify_peer' => false],
    ],
],
```

PhpRedis Serialization and Compression

The `PhpRedis` extension may also be configured to use a variety of serializers and compression algorithms. These algorithms can be configured via the `options` array of your Redis configuration:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'options' => [
        'serializer' => Redis::SERIALIZER_MSGPACK,
        'compression' => Redis::COMPRESSION_LZ4,
    ],
    // Rest of Redis configuration...
],
```

Currently supported serializers include: `Redis::SERIALIZER_NONE` (default), `Redis::SERIALIZER_PHP`, `Redis::SERIALIZER_JSON`, `Redis::SERIALIZER_IGBINARY`, and `Redis::SERIALIZER_MSGPACK`.

Supported compression algorithms include: `Redis::COMPRESSION_NONE` (default), `Redis::COMPRESSION_LZF`, `Redis::COMPRESSION_ZSTD`, and `Redis::COMPRESSION_LZ4`.

Interacting With Redis

You may interact with Redis by calling various methods on the `Redis` facade. The `Redis` facade supports dynamic methods, meaning you may call any [Redis command](#) on the facade and the command will be passed directly to Redis. In this example, we will call the Redis `GET` command by calling the `get` method on the `Redis` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Redis;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => Redis::get('user:profile:'.$id)
        ]);
    }
}
```

As mentioned above, you may call any of Redis' commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server. If a Redis command expects arguments, you should pass those to the facade's corresponding method:

```
use Illuminate\Support\Facades\Redis;

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may pass commands to the server using the `Redis` facade's `command` method, which accepts the name of the command as its first argument and an array of values as its second argument:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Using Multiple Redis Connections

Your application's `config/database.php` configuration file allows you to define multiple Redis connections / servers. You may obtain a connection to a specific Redis connection using the `Redis` facade's `connection` method:

```
$redis = Redis::connection('connection-name');
```

To obtain an instance of the default Redis connection, you may call the `connection` method without any additional arguments:

```
$redis = Redis::connection();
```

Transactions

The `Redis` facade's `transaction` method provides a convenient wrapper around Redis' native `MULTI` and `EXEC` commands. The `transaction` method accepts a closure as its only argument. This closure will receive a Redis connection instance and may issue any commands it would like to this instance. All of the Redis commands issued within the closure will be executed in a single, atomic transaction:

```
use Redis;
use Illuminate\Support\Facades;

\Facades\Redis::transaction(function (Redis $redis) {
    $redis->incr('user_visits', 1);
    $redis->incr('total_visits', 1);
});
```



When defining a Redis transaction, you may not retrieve any values from the Redis connection. Remember, your transaction is executed as a single, atomic operation and that operation is not executed until your entire closure has finished executing its commands.

Lua Scripts

The `eval` method provides another method of executing multiple Redis commands in a single, atomic operation. However, the `eval` method has the benefit of being able to interact with and inspect Redis key values during that operation. Redis scripts are written in the [Lua programming language](#).

The `eval` method can be a bit scary at first, but we'll explore a basic example to break the ice. The `eval` method expects several arguments. First, you should pass the Lua script (as a string) to the method. Secondly, you should pass the number of keys (as an integer) that the script interacts with. Thirdly, you should pass the names of those keys. Finally, you may pass any other additional arguments that you need to access within your script.

In this example, we will increment a counter, inspect its new value, and increment a second counter if the first counter's value is greater than five. Finally, we will return the value of the first counter:

```
$value = Redis::eval(<<<'LUA'
    local counter = redis.call("incr", KEYS[1])

    if counter > 5 then
        redis.call("incr", KEYS[2])
    end

    return counter
LUA, 2, 'first-counter', 'second-counter');
```



Please consult the [Redis documentation](#) for more information on Redis scripting.

Pipelining Commands

Sometimes you may need to execute dozens of Redis commands. Instead of making a network trip to your Redis server for each command, you may use the `pipeline` method. The `pipeline` method accepts one argument: a closure that receives a Redis instance. You may issue all of your commands to this Redis instance and they will all be sent to the Redis server at the same time to reduce network trips to the server. The commands will still be executed in the order they were issued:

```
use Redis;
use Illuminate\Support\Facades;

Facades\Redis::pipeline(function (Redis $pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel provides a convenient interface to the Redis `publish` and `subscribe` commands. These Redis commands allow you to listen for messages on a given "channel". You may publish messages to the channel from another application, or even using another programming language, allowing easy communication between applications and processes.

First, let's setup a channel listener using the `subscribe` method. We'll place this method call within an [Artisan command](#) since calling the `subscribe` method begins a long-running process:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     */
    public function handle(): void
    {
        Redis::subscribe(['test-channel'], function (string $message) {
            echo $message;
        });
    }
}
```

Now we may publish messages to the channel using the `publish` method:

```
use Illuminate\Support\Facades\Redis;

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

    Redis::publish('test-channel', json_encode([
        'name' => 'Adam Wathan'
    ]));
});
```

Wildcard Subscriptions

Using the `psubscribe` method, you may subscribe to a wildcard channel, which may be useful for catching all messages on all channels. The channel name will be passed as the second argument to the provided closure:

```
Redis::psubscribe(['*'], function (string $message, string $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function (string $message, string $channel) {
    echo $message;
});
```

Chapter 8

Eloquent ORM

Eloquent: Getting Started

- [Introduction](#)
- [Generating Model Classes](#)
- [Eloquent Model Conventions](#)
 - [Table Names](#)
 - [Primary Keys](#)
 - [UUID and ULID Keys](#)
 - [Timestamps](#)
 - [Database Connections](#)
 - [Default Attribute Values](#)
 - [Configuring Eloquent Strictness](#)
- [Retrieving Models](#)
 - [Collections](#)
 - [Chunking Results](#)
 - [Chunk Using Lazy Collections](#)
 - [Cursors](#)
 - [Advanced Subqueries](#)
- [Retrieving Single Models / Aggregates](#)
 - [Retrieving or Creating Models](#)
 - [Retrieving Aggregates](#)
- [Inserting and Updating Models](#)
 - [Inserts](#)
 - [Updates](#)
 - [Mass Assignment](#)
 - [Upserts](#)
- [Deleting Models](#)
 - [Soft Deleting](#)
 - [Querying Soft Deleted Models](#)
- [Pruning Models](#)
- [Replicating Models](#)
- [Query Scopes](#)
 - [Global Scopes](#)
 - [Local Scopes](#)
- [Comparing Models](#)
- [Events](#)
 - [Using Closures](#)
 - [Observers](#)
 - [Muting Events](#)

Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.



Before getting started, be sure to configure a database connection in your application's `config/database.php` configuration file. For more information on configuring your database, check out [the database configuration documentation](#).

Laravel Bootcamp

If you're new to Laravel, feel free to jump into the [Laravel Bootcamp](#). The Laravel Bootcamp will walk you through building your first Laravel application using Eloquent. It's a great way to get a tour of everything that Laravel and Eloquent have to offer.

Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the `app\Models` directory and extend the `Illuminate\Database\Eloquent\Model` class. You may use the `make:model` Artisan command to generate a new model:

```
php artisan make:model Flight
```

If you would like to generate a [database migration](#) when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, policies, controllers, and form requests. In addition, these options may be combined to create multiple classes at once:

```
# Generate a model and a FlightFactory class...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generate a model and a FlightSeeder class...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generate a model and a FlightController class...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generate a model, FlightController resource class, and form request classes...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Generate a model and a FlightPolicy class...
php artisan make:model Flight --policy

# Generate a model and a migration, factory, seeder, and controller...
php artisan make:model Flight -mfsc

# Shortcut to generate a model, migration, factory, seeder, policy, controller, and form
# requests...
php artisan make:model Flight --all

# Generate a pivot model...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

Inspecting Models

Sometimes it can be difficult to determine all of a model's available attributes and relationships just by skimming its code. Instead, try the `model:show` Artisan command, which provides a convenient overview of all the model's attributes and relations:

```
php artisan model:show Flight
```

Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    // ...
}
```

Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our `Flight` model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `Flight` model stores records in the `flights` table, while an `AirTrafficController` model would store records in an `air_traffic_controllers` table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a `table` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named `id`. If necessary, you may define a protected `$primaryKey` property on your model to specify a different column that serves as your model's primary key:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to `false`:

```
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of `string`:

```
<?php

class Flight extends Model
{
    /**
     * The data type of the primary key ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

"Composite" Primary Keys

Eloquent requires each model to have at least one uniquely identifying "ID" that can serve as its primary key. "Composite" primary keys are not supported by Eloquent models. However, you are free to add additional multi-column, unique indexes to your database tables in addition to the table's uniquely identifying primary key.

UUID and ULID Keys

Instead of using auto-incrementing integers as your Eloquent model's primary keys, you may choose to use UUIDs instead. UUIDs are universally unique alpha-numeric identifiers that are 36 characters long.

If you would like a model to use a UUID key instead of an auto-incrementing integer key, you may use the `Illuminate\Database\Eloquent\Concerns\HasUuids` trait on the model. Of course, you should ensure that the model has a [UUID equivalent primary key column](#):

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUuids;

    // ...

    $article = Article::create(['title' => 'Traveling to Europe']);

    $article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

By default, The `HasUuids` trait will generate ["ordered" UIDs](#) for your models. These UIDs are more efficient for indexed database storage because they can be sorted lexicographically.

You can override the UUID generation process for a given model by defining a `newUniqueId` method on the model. In addition, you may specify which columns should receive UIDs by defining a `uniqueIds` method on the model:

```
use Ramsey\Uuid\Uuid;

/**
 * Generate a new UUID for the model.
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}

/**
 * Get the columns that should receive a unique identifier.
 *
 * @return array<int, string>
 */
public function uniqueIds(): array
{
    return ['id', 'discount_code'];
}
```

If you wish, you may choose to utilize "ULIDs" instead of UIDs. ULIDs are similar to UIDs; however, they are only 26 characters in length. Like ordered UIDs, ULIDs are lexicographically sortable for efficient database indexing. To utilize ULIDs, you should use the `Illuminate\Database\Eloquent\Concerns\HasUlids` trait on your model. You should also ensure that the model has a [ULID equivalent primary key column](#):

```
use Illuminate\Database\Eloquent\Concerns\HasUlids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUlids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Asia']);

$article->id; // "01gd4d3tgrffqeda94gdbtdk5c"
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

If you would like to perform model operations without the model having its `updated_at` timestamp modified, you may operate on the model within a closure given to the `withoutTimestamps` method:

```
Model::withoutTimestamps(fn () => $post->increment(['reads']));
```

Database Connections

By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The database connection that should be used by the model.
     *
     * @var string
     */
    protected $connection = 'sqlite';
}
```

Default Attribute Values

By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model. Attribute values placed in the `$attributes` array should be in their raw, "storable" format as if they were just read from the database:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

Configuring Eloquent Strictness

Laravel offers several methods that allow you to configure Eloquent's behavior and "strictness" in a variety of situations.

First, the `preventLazyLoading` method accepts an optional boolean argument that indicates if lazy loading should be prevented. For example, you may wish to only disable lazy loading in non-production environments so that your production environment will continue to function normally even if a lazy loaded relationship is accidentally present in production code. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider`:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

Also, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. This can help prevent unexpected errors during local development when attempting to set an attribute that has not been added to the model's `fillable` array:

```
Model::preventSilentlyDiscardingAttributes(! $this->app->isProduction());
```

Retrieving Models

Once you have created a model and `its associated database table`, you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful `query builder` allowing you to fluently query the database table associated with the model. The model's `all` method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Building Queries

The Eloquent `all` method will return all of the results in the model's table. However, since each Eloquent model serves as a `query builder`, you may add additional constraints to queries and then invoke the `get` method to retrieve the results:

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();
```



Since Eloquent models are `query builders`, you should review all of the methods provided by Laravel's `query builder`. You may use any of these methods when writing your Eloquent queries.

Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

The Eloquent `Collection` class extends Laravel's base `Illuminate\Support\Collection` class, which provides a [variety of helpful methods](#) for interacting with data collections. For example, the `reject` method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides [a few extra methods](#) that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the `all` or `get` methods. Instead of using these methods, the `chunk` method may be used to process large numbers of models more efficiently.

The `chunk` method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the `chunk` method will provide significantly reduced memory usage when working with a large number of models:

```
use App\Models\Flight;
use Illuminate\Database\Eloquent\Collection;

Flight::chunk(200, function (Collection $flights) {
    foreach ($flights as $flight) {
        // ...
    }
});
```

The first argument passed to the `chunk` method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the `chunk` method based on a column that you will also be updating while iterating over the results, you should use the `chunkById` method. Using the `chunk` method in these scenarios could lead to unexpected and inconsistent results. Internally, the `chunkById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
->chunkById(200, function (Collection $flights) {
    $flights->each->update(['departed' => false]);
}, $column = 'id');
```

Chunking Using Lazy Collections

The `lazy` method works similarly to the `chunk` method in the sense that, behind the scenes, it executes the query in chunks. However, instead of passing each chunk directly into a callback as is, the `lazy` method returns a flattened `LazyCollection` of Eloquent models, which lets you interact with the results as a single stream:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    // ...
}
```

If you are filtering the results of the `lazy` method based on a column that you will also be updating while iterating over the results, you should use the `lazyById` method. Internally, the `lazyById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
->lazyById(200, $column = 'id')
->each->update(['departed' => false]);
```

You may filter the results based on the descending order of the `id` using the `lazyByIdDesc` method.

Cursors

Similar to the `lazy` method, the `cursor` method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The `cursor` method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor.



Since the `cursor` method only ever holds a single Eloquent model in memory at a time, it cannot eager load relationships. If you need to eager load relationships, consider using the `lazy` method instead.

Internally, the `cursor` method uses PHP [generators](#) to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    // ...
}
```

The `cursor` returns an `Illuminate\Support\LazyCollection` instance. [Lazy collections](#) allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Although the `cursor` method uses far less memory than a regular query (by only holding a single Eloquent model in memory at a time), it will still eventually run out of memory. This is [due to PHP's PDO driver internally caching all raw query results in its buffer](#). If you're dealing with a very large number of Eloquent records, consider using the `lazy` method instead.

Advanced Subqueries

Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight `destinations` and a table of `flights` to destinations. The `flights` table contains an `arrived_at` column which indicates when the flight arrived at the destination.

Using the subquery functionality available to the query builder's `select` and `addSelect` methods, we can select all of the `destinations` and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

Subquery Ordering

In addition, the query builder's `orderBy` function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to perform some other action if no results are found. The `findOr` and `firstOr` methods will return a single model instance or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the method:

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function (string $id) {
    return Flight::findOrFail($id);
});
```

Retrieving or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Retrieving Aggregates

When interacting with Eloquent models, you may also use the `count`, `sum`, `max`, and other [aggregate methods](#) provided by the Laravel [query builder](#). As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Inserting and Updating Models

Inserts

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the `save` method on the model instance:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Store a new flight in the database.
     */
    public function store(Request $request): RedirectResponse
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();

        return redirect('/flights');
    }
}
```

In this example, we assign the `name` field from the incoming HTTP request to the `name` attribute of the `App\Models\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The model's `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Alternatively, you may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the `create` method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the [mass assignment documentation](#).

Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated. The `update` method returns the number of affected rows.



When issuing a mass update via Eloquent, the `saving`, `saved`, `updating`, and `updated` model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

Examining Attribute Changes

Eloquent provides the `isDirty`, `isClean`, and `wasChanged` methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The `isDirty` method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name or an array of attributes to the `isDirty` method to determine if any of the attributes are "dirty". The `isClean` method will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```

use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true

```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

```

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true

```

The `getOriginal` method returns an array containing the original attributes of the model regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

```
$user = User::find(1);

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Array of original attributes...
```

Mass Assignment

You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.

A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Mass Assignment and JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```
/**  
 * The attributes that are mass assignable.  
 *  
 * @var array  
 */  
protected $fillable = [  
    'options->enabled',  
];
```

Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
/**  
 * The attributes that aren't mass assignable.  
 *  
 * @var array  
 */  
protected $guarded = [];
```

Mass Assignment Exceptions

By default, attributes that are not included in the `$fillable` array are silently discarded when performing mass-assignment operations. In production, this is expected behavior; however, during local development it can lead to confusion as to why model changes are not taking effect.

If you wish, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. Typically, this method should be invoked within the `boot` method of one of your application's service providers:

```
use Illuminate\Database\Eloquent\Model;  
  
/**  
 * Bootstrap any application services.  
 */  
public function boot(): void  
{  
    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());  
}
```

Upserts

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the

`save` method.

In the example below, if a flight exists with a `departure` location of `Oakland` and a `destination` location of `San Diego`, its `price` and `discounted` columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

If you would like to perform multiple "upserts" in a single query, then you should use the `upsert` method instead. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The `upsert` method will automatically set the `created_at` and `updated_at` timestamps if timestamps are enabled on the model:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```



All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index. In addition, the MySQL database driver ignores the second argument of the `upsert` method and always uses the "primary" and "unique" indexes of the table to detect existing records.

Deleting Models

To delete a model, you may call the `delete` method on the model instance:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

You may call the `truncate` method to delete all of the model's associated database records. The `truncate` operation will also reset any auto-incrementing IDs on the model's associated table:

```
Flight::truncate();
```

Deleting an Existing Model by its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a [collection](#) of primary keys:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```



The `destroy` method loads each model individually and calls the `delete` method so that the `deleting` and `deleted` events are properly dispatched for each model.

Deleting Models Using Queries

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deleted = Flight::where('active', 0)->delete();
```



When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```



The `SoftDeletes` trait will automatically cast the `deleted_at` attribute to a `DateTime` / `Carbon` instance for you.

You should also add the `deleted_at` column to your database table. The Laravel [schema builder](#) contains a helper method to create this column:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

```
if ($flight->trashed()) {
    // ...
}
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the `restore` method on a model instance. The `restore` method will set the model's `deleted_at` column to `null`:

```
$flight->restore();
```

You may also use the `restore` method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

The `restore` method may also be used when building [relationship](#) queries:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

```
$flight->forceDelete();
```

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

The `withTrashed` method may also be called when building a [relationship](#) query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

Pruning Models

Sometimes you may want to periodically delete models that are no longer needed. To accomplish this, you may add the `Illuminate\Database\Eloquent\Prunable` or `Illuminate\Database\Eloquent\MassPrunable` trait to the models you would like to periodically prune. After adding one of the traits to the model, implement a `prunable` method which returns an Eloquent query builder that resolves the models that are no longer needed:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

When marking models as `Prunable`, you may also define a `pruning` method on the model. This method will be called before the model is deleted. This method can be useful for deleting any additional resources associated with the model, such as stored files, before the model is permanently removed from the database:

```
/**
 * Prepare the model for pruning.
 */
protected function pruning(): void
{
    // ...
}
```

After configuring your prunable model, you should schedule the `model:prune` Artisan command in your application's `App\Console\Kernel` class. You are free to choose the appropriate interval at which this command should be run:

```
/**
 * Define the application's command schedule.
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('model:prune')->daily();
}
```

Behind the scenes, the `model:prune` command will automatically detect "Prunable" models within your application's `app\Models` directory. If your models are in a different location, you may use the `--model` option to specify the model class names:

```
$schedule->command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

If you wish to exclude certain models from being pruned while pruning all other detected models, you may use the `--except` option:

```
$schedule->command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

You may test your `prunable` query by executing the `model:prune` command with the `--pretend` option. When pretending, the `model:prune` command will simply report how many records would be pruned if the command were to actually run:

```
php artisan model:prune --pretend
```



Soft deleting models will be permanently deleted (`forceDelete`) if they match the prunable query.

Mass Pruning

When models are marked with the `Illuminate\Database\Eloquent\MassPrunable` trait, models are deleted from the database using mass-deletion queries. Therefore, the `pruning` method will not be invoked, nor will the `deleting` and `deleted` model events be dispatched. This is because the models are never actually retrieved before deletion, thus making the pruning process much more efficient:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Replicating Models

You may create an unsaved copy of an existing model instance using the `replicate` method. This method is particularly useful when you have model instances that share many of the same attributes:

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

To exclude one or more attributes from being replicated to the new model, you may pass an array to the `replicate` method:

```
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);
```

Query Scopes

Global Scopes

Global scopes allow you to add constraints to all queries for a given model. Laravel's own [soft delete](#) functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

Generating Scopes

To generate a new global scope, you may invoke the `make:scope` Artisan command, which will place the generated scope in your application's `app/Models/Scopes` directory:

```
php artisan make:scope AncientScope
```

Writing Global Scopes

Writing a global scope is simple. First, use the `make:scope` command to generate a class that implements the `Illuminate\Database\Eloquent\Scope` interface. The `Scope` interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints or other types of clauses to the query as needed:

```
<?php

namespace App\Models\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Apply the scope to a given Eloquent query builder.
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```



If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

Applying Global Scopes

To assign a global scope to a model, you may simply place the `ScopedBy` attribute on the model:

```
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Attributes\ScopedBy;

#[ScopedBy( [AncientScope::class] )]
class User extends Model
{
    //
}
```

Or, you may manually register the global scope by overriding the model's `booted` method and invoke the model's `addGlobalScope` method. The `addGlobalScope` method accepts an instance of your scope as its only argument:

```
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::addGlobalScope(new AncientScope());
    }
}
```

After adding the scope in the example above to the `App\Models\User` model, a call to the `User::all()` method will execute the following SQL query:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return the same query builder instance or `void`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

Utilizing a Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the `scope` prefix when calling the method. You can even chain calls to various scopes:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of closures to achieve the correct [logical grouping](#):

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" `orWhere` method that allows you to fluently chain scopes together without the use of closures:

```
$users = User::popular()->orWhere->active()->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     */
    public function scopeOfType(Builder $query, string $type): void
    {
        $query->where('type', $type);
    }
}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

```
$users = User::ofType('admin')->get();
```

Comparing Models

Sometimes you may need to determine if two models are the "same" or not. The `is` and `isNot` methods may be used to quickly verify two models have the same primary key, table, and database connection or not:

```
if ($post->is($anotherPost)) {
    // ...
}

if ($post->isNot($anotherPost)) {
    // ...
}
```

The `is` and `isNot` methods are also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` [relationships](#). This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

```
if ($post->author()->is($user)) {
    // ...
}
```

Events



Want to broadcast your Eloquent events directly to your client-side application? Check out Laravel's [model event broadcasting](#).

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle:

`retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleting`, `forceDeleted`, `restoring`, `restored`, and `replicating`.

The `retrieved` event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will dispatch. The `updating` / `updated` events will dispatch when an existing model is modified and the `save` method is called. The `saving` / `saved` events will dispatch when a model is created or updated – even if the model's attributes have not been changed. Event names ending with `-ing` are dispatched before any changes to the model are persisted, while events ending with `-ed` are dispatched after the changes to the model are persisted.

To start listening to model events, define a `$dispatchesEvents` property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own [event classes](#). Each model event class should expect to receive an instance of the affected model via its constructor:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

After defining and mapping your Eloquent events, you may use [event listeners](#) to handle the events.



When issuing a mass update or delete query via Eloquent, the `saved`, `updated`, `deleting`, and `deleted` model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing mass updates or deletes.

Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the `booted` method of your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::created(function (User $user) {
            // ...
        });
    }
}
```

If needed, you may utilize [queueable anonymous event listeners](#) when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's [queue](#):

```
use function Illuminate\Events\queueable;

static::created(queueable(function (User $user) {
    // ...
}));
```

Observers

Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as their only argument. The `make:observer` Artisan command is the easiest way to create a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `app/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Handle the User "created" event.
     */
    public function created(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "updated" event.
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "deleted" event.
     */
    public function deleted(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "restored" event.
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "forceDeleted" event.
     */
    public function forceDeleted(User $user): void
    {
        // ...
    }
}
```

To register an observer, you may place the `ObservedBy` attribute on the corresponding model:

```
use App\Observers\UserObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

#[ObservedBy( [UserObserver::class] )]
class User extends Authenticatable
{
    //
}
```

Or, you may manually register an observer by calling the `observe` method on the model you wish to observe. You may register observers in the `boot` method of your application's `App\Providers\EventServiceProvider` service provider:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Register any events for your application.
 */
public function boot(): void
{
    User::observe(UserObserver::class);
}
```



There are additional events an observer can listen to, such as `saving` and `retrieved`. These events are described within the [events](#) documentation.

Observers and Database Transactions

When models are being created within a database transaction, you may want to instruct an observer to only execute its event handlers after the database transaction is committed. You may accomplish this by implementing the `ShouldHandleEventsAfterCommit` interface on your observer. If a database transaction is not in progress, the event handlers will execute immediately:

```
<?php

namespace App\Observers;

use App\Models\User;
use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;

class UserObserver implements ShouldHandleEventsAfterCommit
{
    /**
     * Handle the User "created" event.
     */
    public function created(User $user): void
    {
        // ...
    }
}
```

Muting Events

You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events, and any value returned by the closure will be returned by the `withoutEvents` method:

```
use App\Models\User;

$user = User::withoutEvents(function () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Saving a Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

You may also "update", "delete", "soft delete", "restore", and "replicate" a given model without dispatching any events:

```
$user->deleteQuietly();
$user->forceDeleteQuietly();
$user->restoreQuietly();
```

Eloquent: Relationships

- [Introduction](#)
- [Defining Relationships](#)
 - [One to One](#)
 - [One to Many](#)
 - [One to Many \(Inverse\) / Belongs To](#)
 - [Has One of Many](#)
 - [Has One Through](#)
 - [Has Many Through](#)
- [Many to Many Relationships](#)
 - [Retrieving Intermediate Table Columns](#)
 - [Filtering Queries via Intermediate Table Columns](#)
 - [Ordering Queries via Intermediate Table Columns](#)
 - [Defining Custom Intermediate Table Models](#)
- [Polymorphic Relationships](#)
 - [One to One](#)
 - [One to Many](#)
 - [One of Many](#)
 - [Many to Many](#)
 - [Custom Polymorphic Types](#)
- [Dynamic Relationships](#)
- [Querying Relations](#)
 - [Relationship Methods vs. Dynamic Properties](#)
 - [Querying Relationship Existence](#)
 - [Querying Relationship Absence](#)
 - [Querying Morph To Relationships](#)
- [Aggregating Related Models](#)
 - [Counting Related Models](#)
 - [Other Aggregate Functions](#)
 - [Counting Related Models on Morph To Relationships](#)
- [Eager Loading](#)
 - [Constraining Eager Loads](#)
 - [Lazy Eager Loading](#)
 - [Preventing Lazy Loading](#)
- [Inserting and Updating Related Models](#)
 - [The `save` Method](#)
 - [The `create` Method](#)
 - [Belongs To Relationships](#)
 - [Many to Many Relationships](#)
- [Touching Parent Timestamps](#)

Introduction

Database tables are often related to one another. For example, a blog post may have many comments or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports a variety of common relationships:

- [One To One](#)
- [One To Many](#)
- [Many To Many](#)

- [Has One Through](#)
- [Has Many Through](#)
- [One To One \(Polymorphic\)](#)
- [One To Many \(Polymorphic\)](#)
- [Many To Many \(Polymorphic\)](#)

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as powerful [query builders](#), defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this `posts` relationship:

```
$user->posts()->where('active', 1)->get();
```

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

One to One

A one-to-one relationship is a very basic type of database relationship. For example, a `User` model might be associated with one `Phone` model. To define this relationship, we will place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result. The `hasOne` method is available to your model via the model's `Illuminate\Database\Eloquent\Model` base class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the parent model name. In this case, the `Phone` model is automatically assumed to have a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the primary key column of the parent. In other words, Eloquent will look for the value of the user's `id` column in the `user_id` column of the `Phone` record. If you would like the relationship to use a primary key value other than `id` or your model's `$primaryKey` property, you may pass a third argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

Defining the Inverse of the Relationship

So, we can access the `Phone` model from our `User` model. Next, let's define a relationship on the `Phone` model that will let us access the user that owns the phone. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

When invoking the `user` method, Eloquent will attempt to find a `User` model that has an `id` which matches the `user_id` column on the `Phone` model.

Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. So, in this case, Eloquent assumes that the `Phone` model has a `user_id` column. However, if the foreign key on the `Phone` model is not `user_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the user that owns the phone.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

If the parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying the parent table's custom key:

```
/**
 * Get the user that owns the phone.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

One to Many

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column for the `Comment` model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with `_id`. So, in this example, Eloquent will assume the foreign key column on the `Comment` model is `post_id`.

Once the relationship method has been defined, we can access the `collection` of related comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were defined as properties on the model:

```
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    // ...
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = Post::find(1)->comments()
    ->where('title', 'foo')
    ->first();
```

Like the `hasOne` method, you may also override the foreign and local keys by passing additional arguments to the `hasMany` method:

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

One to Many (Inverse) / Belongs To

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a `hasMany` relationship, define a relationship method on the child model which calls the `belongsTo` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the `post` "dynamic relationship property":

```
use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;
```

In the example above, Eloquent will attempt to find a `Post` model that has an `id` which matches the `post_id` column on the `Comment` model.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a `_` followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the `Post` model's foreign key on the `comments` table is `post_id`.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the `belongsTo` method:

```
/**
 * Get the post that owns the comment.
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the post that owns the comment.
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Default Models

The `belongsTo`, `hasOne`, `hasOneThrough`, and `morphOne` relationships allow you to define a default model that will be returned if the given relationship is `null`. This pattern is often referred to as the [Null Object pattern](#) and can help remove conditional checks in your code. In the following example, the `user` relation will return an empty `App\Models\User` model if no user is attached to the `Post` model:

```
/**
 * Get the author of the post.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault();
}
```

To populate the default model with attributes, you may pass an array or closure to the `withDefault` method:

```

/**
 * Get the author of the post.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault(function (User $user, Post $post) {
        $user->name = 'Guest Author';
    });
}

```

Querying Belongs To Relationships

When querying for the children of a "belongs to" relationship, you may manually build the `where` clause to retrieve the corresponding Eloquent models:

```

use App\Models\Post;

$posts = Post::where('user_id', $user->id)->get();

```

However, you may find it more convenient to use the `whereBelongsTo` method, which will automatically determine the proper relationship and foreign key for the given model:

```
$posts = Post::whereBelongsTo($user)->get();
```

You may also provide a [collection](#) instance to the `whereBelongsTo` method. When doing so, Laravel will retrieve models that belong to any of the parent models within the collection:

```

$users = User::where('vip', true)->get();

$posts = Post::whereBelongsTo($users)->get();

```

By default, Laravel will determine the relationship associated with the given model based on the class name of the model; however, you may specify the relationship name manually by providing it as the second argument to the `whereBelongsTo` method:

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

Has One of Many

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Order` models, but you want to define a convenient way to interact with the most recent order the user has placed. You may accomplish this using the `hasOne` relationship type combined with the `ofMany` methods:

```
/**
 * Get the user's most recent order.
 */
public function latestOrder(): HasOne
{
    return $this->hasOne(Order::class)->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```
/**
 * Get the user's oldest order.
 */
public function oldestOrder(): HasOne
{
    return $this->hasOne(Order::class)->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most expensive order. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```
/**
 * Get the user's largest order.
 */
public function largestOrder(): HasOne
{
    return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```



Because PostgreSQL does not support executing the `MAX` function against UUID columns, it is not currently possible to use one-of-many relationships in combination with PostgreSQL UUID columns.

Converting "Many" Relationships to Has One Relationships

Often, when retrieving a single model using the `latestOfMany`, `oldestOfMany`, or `ofMany` methods, you already have a "has many" relationship defined for the same model. For convenience, Laravel allows you to easily convert this relationship into a "has one" relationship by invoking the `one` method on the relationship:

```
/**
 * Get the user's orders.
 */
public function orders(): HasMany
{
    return $this->hasMany(Order::class);
}

/**
 * Get the user's largest order.
 */
public function largestOrder(): HasOne
{
    return $this->orders()->one()->ofMany('price', 'max');
}
```

Advanced Has One of Many Relationships

It is possible to construct more advanced "has one of many" relationships. For example, a `Product` model may have many associated `Price` models that are retained in the system even after new pricing is published. In addition, new pricing data for the product may be able to be published in advance to take effect at a future date via a `published_at` column.

So, in summary, we need to retrieve the latest published pricing where the published date is not in the future. In addition, if two prices have the same published date, we will prefer the price with the greatest ID. To accomplish this, we must pass an array to the `ofMany` method that contains the sortable columns which determine the latest price. In addition, a closure will be provided as the second argument to the `ofMany` method. This closure will be responsible for adding additional publish date constraints to the relationship query:

```
/**
 * Get the current pricing for the product.
 */
public function currentPricing(): HasOne
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function (Builder $query) {
        $query->where('published_at', '<', now());
    });
}
```

Has One Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

For example, in a vehicle repair shop application, each `Mechanic` model may be associated with one `Car` model, and each `Car` model may be associated with one `Owner` model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner *through* the `Car` model. Let's look at the tables necessary to define this relationship:

```

mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer

```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Mechanic` model:

```

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOneThrough;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}

```

The first argument passed to the `hasOneThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Or, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-one-through" relationship by invoking the `through` method and supplying the names of those relationships. For example, if the `Mechanic` model has a `cars` relationship and the `Car` model has an `owner` relationship, you may define a "has-one-through" relationship connecting the mechanic and the owner like so:

```

// String based syntax...
return $this->through('cars')->has('owner');

// Dynamic syntax...
return $this->throughCars()->hasOwner();

```

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasOneThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars table...
            'car_id', // Foreign key on the owners table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}

```

Or, as discussed earlier, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-one-through" relationship by invoking the `through` method and supplying the names of those relationships. This approach offers the advantage of reusing the key conventions already defined on the existing relationships:

```

// String based syntax...
return $this->through('cars')->has('owner');

// Dynamic syntax...
return $this->throughCars()->hasOwner();

```

Has Many Through

The "has-many-through" relationship provides a convenient way to access distant relations via an intermediate relation. For example, let's assume we are building a deployment platform like [Laravel Vapor](#). A `Project` model might access many `Deployment` models through an intermediate `Environment` model. Using this example, you could easily gather all deployments for a given project. Let's look at the tables required to define this relationship:

```

projects
id - integer
name - string

environments
id - integer
project_id - integer
name - string

deployments
id - integer
environment_id - integer
commit_hash - string

```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Project` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;

class Project extends Model
{
    /**
     * Get all of the deployments for the project.
     */
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(Deployment::class, Environment::class);
    }
}
```

The first argument passed to the `hasManyThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Or, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-many-through" relationship by invoking the `through` method and supplying the names of those relationships. For example, if the `Project` model has a `environments` relationship and the `Environment` model has a `deployments` relationship, you may define a "has-many-through" relationship connecting the project and the deployments like so:

```
// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();
```

Though the `Deployment` model's table does not contain a `project_id` column, the `hasManyThrough` relation provides access to a project's deployments via `$project->deployments`. To retrieve these models, Eloquent inspects the `project_id` column on the intermediate `Environment` model's table. After finding the relevant environment IDs, they are used to query the `Deployment` model's table.

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```

class Project extends Model
{
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Foreign key on the environments table...
            'environment_id', // Foreign key on the deployments table...
            'id', // Local key on the projects table...
            'id' // Local key on the environments table...
        );
    }
}

```

Or, as discussed earlier, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-many-through" relationship by invoking the `through` method and supplying the names of those relationships. This approach offers the advantage of reusing the key conventions already defined on the existing relationships:

```

// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();

```

Many to Many Relationships

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

Table Structure

To define this relationship, three database tables are needed: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names and contains `user_id` and `role_id` columns. This table is used as an intermediate table linking the users and roles.

Remember, since a role can belong to many users, we cannot simply place a `user_id` column on the `roles` table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the `role_user` table is needed. We can summarize the relationship's table structure like so:

```

users
id - integer
name - string

roles
id - integer
name - string

role_user
user_id - integer
role_id - integer

```

Model Structure

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. The `belongsToMany` method is provided by the `Illuminate\Database\Eloquent\Model` base class that is used by all of your application's Eloquent models. For example, let's define a `roles` method on our `User` model. The first argument passed to this method is the name of the related model class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic relationship property:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    // ...
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `roles` method and continuing to chain conditions onto the query:

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

To determine the table name of the relationship's intermediate table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the `belongsToMany` method:

```
return $this->belongsToMany(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate table, you may also customize the column names of the keys on the table by passing additional arguments to the `belongsToMany` method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

Defining the Inverse of the Relationship

To define the "inverse" of a many-to-many relationship, you should define a method on the related model which also returns the result of the `belongsToMany` method. To complete our user / role example, let's define the `users` method on the `Role` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` model counterpart with the exception of referencing the `App\Models\User` model. Since we're reusing the `belongsToMany` method, all of the usual table and key customization options are available when defining the "inverse" of many-to-many relationships.

Retrieving Intermediate Table Columns

As you have already learned, working with many-to-many relations requires the presence of an intermediate table. Eloquent provides some very helpful ways of interacting with this table. For example, let's assume our `User` model has many `Role` models that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notice that each `Role` model we retrieve is automatically assigned a `pivot` attribute. This attribute contains a model representing the intermediate table.

By default, only the model keys will be present on the `pivot` model. If your intermediate table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

If you would like your intermediate table to have `created_at` and `updated_at` timestamps that are automatically maintained by Eloquent, call the `withTimestamps` method when defining the relationship:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```



Intermediate tables that utilize Eloquent's automatically maintained timestamps are required to have both `created_at` and `updated_at` timestamp columns.

Customizing the `pivot` Attribute Name

As noted previously, attributes from the intermediate table may be accessed on models via the `pivot` attribute. However, you are free to customize the name of this attribute to better reflect its purpose within your application.

For example, if your application contains users that may subscribe to podcasts, you likely have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table attribute to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany(Podcast::class)
    ->as('subscription')
    ->withTimestamps();
```

Once the custom intermediate table attribute has been specified, you may access the intermediate table data using the customized name:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

Filtering Queries via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` relationship queries using the `wherePivot`, `wherePivotIn`, `wherePivotNotIn`, `wherePivotBetween`, `wherePivotNotBetween`, `wherePivotNull`, and `wherePivotNotNull` methods when defining the relationship:

```

return $this->belongsToMany(Role::class)
    ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
    ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
    ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31
00:00:00']);

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31
00:00:00']);

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNotNull('expired_at');

```

Ordering Queries via Intermediate Table Columns

You can order the results returned by `belongsToMany` relationship queries using the `orderByPivot` method. In the following example, we will retrieve all of the latest badges for the user:

```

return $this->belongsToMany(Badge::class)
    ->where('rank', 'gold')
    ->orderByPivot('created_at', 'desc');

```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your many-to-many relationship, you may call the `using` method when defining the relationship. Custom pivot models give you the opportunity to define additional behavior on the pivot model, such as methods and casts.

Custom many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class while custom polymorphic many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\MorphPivot` class. For example, we may define a `Role` model which uses a custom `RoleUser` pivot model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

When defining the `RoleUser` model, you should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    // ...
}
```



Pivot models may not use the `SoftDeletes` trait. If you need to soft delete pivot records consider converting your pivot model to an actual Eloquent model.

Custom Pivot Models and Incrementing IDs

If you have defined a many-to-many relationship that uses a custom pivot model, and that pivot model has an auto-incrementing primary key, you should ensure your custom pivot model class defines an `incrementing` property that is set to `true`.

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

Polymorphic Relationships

A polymorphic relationship allows the child model to belong to more than one type of model using a single association. For example, imagine you are building an application that allows users to share blog posts and videos. In such an application, a `Comment` model might belong to both the `Post` and `Video` models.

One to One (Polymorphic)

Table Structure

A one-to-one polymorphic relation is similar to a typical one-to-one relation; however, the child model can belong to more than one type of model using a single association. For example, a blog `Post` and a `User` may share a polymorphic relation to an `Image` model. Using a one-to-one polymorphic relation allows you to have a single table of unique images that may be associated with posts and users. First, let's examine the table structure:

```
posts
id - integer
name - string

users
id - integer
name - string

images
id - integer
url - string
imageable_id - integer
imageable_type - string
```

Note the `imageable_id` and `imageable_type` columns on the `images` table. The `imageable_id` column will contain the ID value of the post or user, while the `imageable_type` column will contain the class name of the parent model. The `imageable_type` column is used by Eloquent to determine which "type" of parent model to return when accessing the `imageable` relation. In this case, the column would contain either `App\Models\Post` or `App\Models\User`.

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Image extends Model
{
    /**
     * Get the parent imageable model (user or post).
     */
    public function imageable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

Retrieving the Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to retrieve the image for a post, we can access the `image` dynamic relationship property:

```
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

You may retrieve the parent of the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `imageable` method on the `Image` model. So, we will access that method as a dynamic relationship property:

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

The `imageable` relation on the `Image` model will return either a `Post` or `User` instance, depending on which type of model owns the image.

Key Conventions

If necessary, you may specify the name of the "id" and "type" columns utilized by your polymorphic child model. If you do so, ensure that you always pass the name of the relationship as the first argument to the `morphTo` method. Typically, this value should match the method name, so you may use PHP's `__FUNCTION__` constant:

```
/**
 * Get the model that the image belongs to.
 */
public function imageable(): MorphTo
{
    return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

One to Many (Polymorphic)

Table Structure

A one-to-many polymorphic relation is similar to a typical one-to-many relation; however, the child model can belong to more than one type of model using a single association. For example, imagine users of your application can "comment" on posts and videos. Using polymorphic relationships, you may use a single `comments` table to contain comments for both posts and videos. First, let's examine the table structure required to build this relationship:

```
posts
  id - integer
  title - string
  body - text

videos
  id - integer
  title - string
  url - string

comments
  id - integer
  body - text
  commentable_id - integer
  commentable_type - string
```

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).
     */
    public function commentable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

```

Retrieving the Relationship

Once your database table and models are defined, you may access the relationships via your model's dynamic relationship properties. For example, to access all of the comments for a post, we can use the `comments` dynamic property:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    // ...
}
```

You may also retrieve the parent of a polymorphic child model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `commentable` method on the `Comment` model. So, we will access that method as a dynamic relationship property in order to access the comment's parent model:

```
use App\Models\Comment;

$comment = Comment::find(1);

$commentable = $comment->commentable;
```

The `commentable` relation on the `Comment` model will return either a `Post` or `Video` instance, depending on which type of model is the comment's parent.

One of Many (Polymorphic)

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Image` models, but you want to define a convenient way to interact with the most recent image the user has uploaded. You may accomplish this using the `morphOne` relationship type combined with the `ofMany` methods:

```
/**
 * Get the user's most recent image.
 */
public function latestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```
/**
 * Get the user's oldest image.
 */
public function oldestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most "liked" image. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```
/**
 * Get the user's most popular image.
 */
public function bestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->ofMany('likes', 'max');
}
```



It is possible to construct more advanced "one of many" relationships. For more information, please consult the [has one of many documentation](#).

Many to Many (Polymorphic)

Table Structure

Many-to-many polymorphic relations are slightly more complicated than "morph one" and "morph many" relationships. For example, a `Post` model and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation in this situation would allow your application to have a single table of unique tags that may be associated with posts or videos. First, let's examine the table structure required to build this relationship:

```
posts
id - integer
name - string

videos
id - integer
name - string

tags
id - integer
name - string

taggables
tag_id - integer
taggable_id - integer
taggable_type - string
```



Before diving into polymorphic many-to-many relationships, you may benefit from reading the documentation on typical [many-to-many relationships](#).

Model Structure

Next, we're ready to define the relationships on the models. The `Post` and `Video` models will both contain a `tags` method that calls the `morphToMany` method provided by the base Eloquent model class.

The `morphToMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags(): MorphToMany
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

Defining the Inverse of the Relationship

Next, on the `Tag` model, you should define a method for each of its possible parent models. So, in this example, we will define a `posts` method and a `videos` method. Both of these methods should return the result of the `morphedByMany` method.

The `morphedByMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts(): MorphToMany
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos(): MorphToMany
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Retrieving the Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the tags for a post, you may use the `tags` dynamic relationship property:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->tags as $tag) {
    // ...
}
```

You may retrieve the parent of a polymorphic relation from the polymorphic child model by accessing the name of the method that performs the call to `morphedByMany`. In this case, that is the `posts` or `videos` methods on the `Tag` model:

```
use App\Models\Tag;

$tag = Tag::find(1);

foreach ($tag->posts as $post) {
    // ...
}

foreach ($tag->videos as $video) {
    // ...
}
```

Custom Polymorphic Types

By default, Laravel will use the fully qualified class name to store the "type" of the related model. For instance, given the one-to-many relationship example above where a `Comment` model may belong to a `Post` or a `Video` model, the default `commentable_type` would be either `App\Models\Post` or `App\Models\Video`, respectively. However, you may wish to decouple these values from your application's internal structure.

For example, instead of using the model names as the "type", we may use simple strings such as `post` and `video`. By doing so, the polymorphic "type" column values in our database will remain valid even if the models are renamed:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);
```

You may call the `enforceMorphMap` method in the `boot` method of your `App\Providers\AppServiceProvider` class or create a separate service provider if you wish.

You may determine the morph alias of a given model at runtime using the model's `getMorphClass` method. Conversely, you may determine the fully-qualified class name associated with a morph alias using the `Relation::getMorphedModel` method:

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```



*When adding a "morph map" to your existing application, every morphable `*_type` column value in your database that still contains a fully-qualified class will need to be converted to its "map" name.*

Dynamic Relationships

You may use the `resolveRelationUsing` method to define relations between Eloquent models at runtime. While not typically recommended for normal application development, this may occasionally be useful when developing Laravel packages.

The `resolveRelationUsing` method accepts the desired relationship name as its first argument. The second argument passed to the method should be a closure that accepts the model instance and returns a valid Eloquent relationship definition. Typically, you should configure dynamic relationships within the boot method of a [service provider](#):

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function (Order $orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```



When defining dynamic relationships, always provide explicit key name arguments to the Eloquent relationship methods.

Querying Relations

Since all Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing a query to load the related models. In addition, all types of Eloquent relationships also serve as [query builders](#), allowing you to continue to chain constraints onto the relationship query before finally executing the SQL query against your database.

For example, imagine a blog application in which a `User` model has many associated `Post` models:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts(): HasMany
    {
        return $this->hasMany(Post::class);
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

```
use App\Models\User;

$user = User::find(1);

$user->posts()->where('active', 1)->get();
```

You are able to use any of the Laravel [query builder's](#) methods on the relationship, so be sure to explore the query builder documentation to learn about all of the methods that are available to you.

Chaining `orWhere` Clauses After Relationships

As demonstrated in the example above, you are free to add additional constraints to relationships when querying them. However, use caution when chaining `orWhere` clauses onto a relationship, as the `orWhere` clauses will be logically grouped at the same level as the relationship constraint:

```
$user->posts()
    ->where('active', 1)
    ->orWhere('votes', '>=', 100)
    ->get();
```

The example above will generate the following SQL. As you can see, the `or` clause instructs the query to return *any* post with greater than 100 votes. The query is no longer constrained to a specific user:

```
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

In most situations, you should use [logical groups](#) to group the conditional checks between parentheses:

```
use Illuminate\Database\Eloquent\Builder;

$user->posts()
    ->where(function (Builder $query) {
        return $query->where('active', 1)
            ->orWhere('votes', '>=', 100);
    })
->get();
```

The example above will produce the following SQL. Note that the logical grouping has properly grouped the constraints and the query remains constrained to a specific user:

```
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

Relationship Methods vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may access the relationship as if it were a property. For example, continuing to use our `User` and `Post` example models, we may access all of a user's posts like so:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->posts as $post) {
    // ...
}
```

Dynamic relationship properties perform "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use [eager loading](#) to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

Querying Relationship Existence

When retrieving model records, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the `has` and `orHas` methods:

```
use App\Models\Post;

// Retrieve all posts that have at least one comment...
$posts = Post::has('comments')->get();
```

You may also specify an operator and count value to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment that has at least one image:

```
// Retrieve posts that have at least one comment with images...
$posts = Post::has('comments.images')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to define additional query constraints on your `has` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like code...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like code...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```



Eloquent does not currently support querying for relationship existence across databases. The relationships must exist within the same database.

Inline Relationship Existence Queries

If you would like to query for a relationship's existence with a single, simple where condition attached to the relationship query, you may find it more convenient to use the `whereRelation`, `orWhereRelation`, `whereMorphRelation`, and `orWhereMorphRelation` methods. For example, we may query for all posts that have unapproved comments:

```
use App\Models\Post;

$posts = Post::whereRelation('comments', 'is_approved', false)->get();
```

Of course, like calls to the query builder's `where` method, you may also specify an operator:

```
$posts = Post::whereRelation(
    'comments', 'created_at', '>=', now()->subHour()
)->get();
```

Querying Relationship Absence

When retrieving model records, you may wish to limit your results based on the absence of a relationship. For example, imagine you want to retrieve all blog posts that **don't** have any comments. To do so, you may pass the name of the relationship to the `doesntHave` and `orWhereDoesntHave` methods:

```
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to add additional query constraints to your `doesntHave` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

You may use "dot" notation to execute a query against a nested relationship. For example, the following query will retrieve all posts that do not have comments; however, posts that have comments from authors that are not banned will be included in the results:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder $query) {
    $query->where('banned', 0);
})->get();
```

Querying Morph To Relationships

To query the existence of "morph to" relationships, you may use the `whereHasMorph` and `whereDoesntHaveMorph` methods. These methods accept the name of the relationship as their first argument. Next, the methods accept the names of the related models that you wish to include in the query. Finally, you may provide a closure which customizes the relationship query:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// Retrieve comments associated to posts or videos with a title like code...
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// Retrieve comments associated to posts with a title not like code...
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class,
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();
```

You may occasionally need to add query constraints based on the "type" of the related polymorphic model. The closure passed to the `whereHasMorph` method may receive a `$type` value as its second argument. This argument allows you to inspect the "type" of the query that is being built:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, string $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

Querying All Related Models

Instead of passing an array of possible polymorphic models, you may provide `*` as a wildcard value. This will instruct Laravel to retrieve all of the possible polymorphic types from the database. Laravel will execute an additional query in order to perform this operation:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query) {
    $query->where('title', 'like', 'foo%');
})->get();
```

Aggregating Related Models

Counting Related Models

Sometimes you may want to count the number of related models for a given relationship without actually loading the models. To accomplish this, you may use the `withCount` method. The `withCount` method will place a `{relation}_count` attribute on the resulting models:

```
use App\Models\Post;

$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

By passing an array to the `withCount` method, you may add the "counts" for multiple relations as well as add additional constraints to the queries:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount(['votes', 'comments' => function (Builder $query) {
    $query->where('content', 'like', 'code%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

You may also alias the relationship count result, allowing multiple counts on the same relationship:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
])->get();

echo $posts[0]->comments_count;
echo $posts[0]->pending_comments_count;
```

Deferred Count Loading

Using the `loadCount` method, you may load a relationship count after the parent model has already been retrieved:

```
$book = Book::first();

$book->loadCount('genres');
```

If you need to set additional query constraints on the count query, you may pass an array keyed by the relationships you wish to count. The array values should be closures which receive the query builder instance:

```
$book->loadCount(['reviews' => function (Builder $query) {
    $query->where('rating', 5);
}]);
```

Relationship Counting and Custom Select Statements

If you're combining `withCount` with a `select` statement, ensure that you call `withCount` after the `select` method:

```
$posts = Post::select(['title', 'body'])
    ->withCount('comments')
    ->get();
```

Other Aggregate Functions

In addition to the `withCount` method, Eloquent provides `withMin`, `withMax`, `withAvg`, `withSum`, and `withExists` methods. These methods will place a `{relation}_{function}_{column}` attribute on your resulting models:

```
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

If you wish to access the result of the aggregate function using another name, you may specify your own alias:

```
$posts = Post::withSum('comments as total_comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->total_comments;
}
```

Like the `loadCount` method, deferred versions of these methods are also available. These additional aggregate operations may be performed on Eloquent models that have already been retrieved:

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

If you're combining these aggregate methods with a `select` statement, ensure that you call the aggregate methods after the `select` method:

```
$posts = Post::select(['title', 'body'])
    ->withExists('comments')
    ->get();
```

Counting Related Models on Morph To Relationships

If you would like to eager load a "morph to" relationship, as well as related model counts for the various entities that may be returned by that relationship, you may utilize the `with` method in combination with the `morphTo` relationship's `morphWithCount` method.

In this example, let's assume that `Photo` and `Post` models may create `ActivityFeed` models. We will assume the `ActivityFeed` model defines a "morph to" relationship named `parentable` that allows us to retrieve the parent `Photo` or `Post` model for a given `ActivityFeed` instance. Additionally, let's assume that `Photo` models "have many" `Tag` models and `Post` models "have many" `Comment` models.

Now, let's imagine we want to retrieve `ActivityFeed` instances and eager load the `parentable` parent models for each `ActivityFeed` instance. In addition, we want to retrieve the number of tags that are associated with each parent photo and the number of comments that are associated with each parent post:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
    'parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWithCount([
            Photo::class => ['tags'],
            Post::class => ['comments'],
        ]);
    }
])->get();
```

Deferred Count Loading

Let's assume we have already retrieved a set of `ActivityFeed` models and now we would like to load the nested relationship counts for the various `parentable` models associated with the activity feeds. You may use the `loadMorphCount` method to accomplish this:

```
$activities = ActivityFeed::with('parentable')->get();

$activities->loadMorphCount('parentable', [
    Photo::class => ['tags'],
    Post::class => ['comments'],
]);
```

Eager Loading

When accessing Eloquent relationships as properties, the related models are "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the "N + 1" query problem. To illustrate the N + 1 query problem, consider a `Book` model that "belongs to" to an `Author` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author(): BelongsTo
    {
        return $this->belongsTo(Author::class);
    }
}
```

Now, let's retrieve all books and their authors:

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute one query to retrieve all of the books within the database table, then another query for each book in order to retrieve the book's author. So, if we have 25 books, the code above would run 26 queries: one for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just two queries. When building a query, you may specify which relationships should be eager loaded using the `with` method:

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed - one query to retrieve all of the books and one query to retrieve all of the authors for all of the books:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships. To do so, just pass an array of relationships to the `with` method:

```
$books = Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager load a relationship's relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts:

```
$books = Book::with('author.contacts')->get();
```

Alternatively, you may specify nested eager loaded relationships by providing a nested array to the `with` method, which can be convenient when eager loading multiple nested relationships:

```
$books = Book::with([
    'author' => [
        'contacts',
        'publisher',
    ],
])->get();
```

Nested Eager Loading `morphTo` Relationships

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `with` method in combination with the `morphTo` relationship's `morphWith` method. To help illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()
    ->with(['parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWith([
            Event::class => ['calendar'],
            Photo::class => ['tags'],
            Post::class => ['author'],
        ]);
    }])->get();
```

Eager Loading Specific Columns

You may not always need every column from the relationships you are retrieving. For this reason, Eloquent allows you to specify which columns of the relationship you would like to retrieve:

```
$books = Book::with('author:id,name,book_id')->get();
```



When using this feature, you should always include the `id` column and any relevant foreign key columns in the list of columns you wish to retrieve.

Eager Loading by Default

Sometimes you might want to always load some relationships when retrieving a model. To accomplish this, you may define a `$with` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
    /**
     * The relationships that should always be loaded.
     *
     * @var array
     */
    protected $with = ['author'];

    /**
     * Get the author that wrote the book.
     */
    public function author(): BelongsTo
    {
        return $this->belongsTo(Author::class);
    }

    /**
     * Get the genre of the book.
     */
    public function genre(): BelongsTo
    {
        return $this->belongsTo(Genre::class);
    }
}
```

If you would like to remove an item from the `$with` property for a single query, you may use the `without` method:

```
$books = Book::without('author')->get();
```

If you would like to override all items within the `$with` property for a single query, you may use the `withOnly` method:

```
$books = Book::withOnly('genre')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship but also specify additional query conditions for the eager loading query. You can accomplish this by passing an array of relationships to the `with` method where the array key is a relationship name and the array value is a closure that adds additional constraints to the eager loading query:

```
use App\Models\User;
use Illuminate\Contracts\Database\Eloquent\Builder;

$users = User::with(['posts' => function (Builder $query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `code`. You may call other `query builder` methods to further customize the eager loading operation:

```
$users = User::with(['posts' => function (Builder $query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```



The `limit` and `take` query builder methods may not be used when constraining eager loads.

Constraining Eager Loading of `morphTo` Relationships

If you are eager loading a `morphTo` relationship, Eloquent will run multiple queries to fetch each type of related model. You may add additional constraints to each of these queries using the `MorphTo` relation's `constrain` method:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
    $morphTo->constrain([
        Post::class => function ($query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function ($query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

In this example, Eloquent will only eager load posts that have not been hidden and videos that have a `type` value of "educational".

Constraining Eager Loads With Relationship Existence

You may sometimes find yourself needing to check for the existence of a relationship while simultaneously loading the relationship based on the same conditions. For example, you may wish to only retrieve `User` models that have child `Post` models matching a given query condition while also eager loading the matching posts. You may accomplish this using the `withWhereHas` method:

```
use App\Models\User;

$users = User::withWhereHas('posts', function ($query) {
    $query->where('featured', true);
})->get();
```

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

```
use App\Models\Book;

$books = Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

If you need to set additional query constraints on the eager loading query, you may pass an array keyed by the relationships you wish to load. The array values should be closure instances which receive the query instance:

```
$author->load(['books' => function (Builder $query) {
    $query->orderBy('published_date', 'asc');
}]);
```

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```
$book->loadMissing('author');
```

Nested Lazy Eager Loading and `morphTo`

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `loadMorph` method.

This method accepts the name of the `morphTo` relationship as its first argument, and an array of model / relationship pairs as its second argument. To help illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
});
```

Preventing Lazy Loading

As previously discussed, eager loading relationships can often provide significant performance benefits to your application. Therefore, if you would like, you may instruct Laravel to always prevent the lazy loading of relationships. To accomplish this, you may invoke the `preventLazyLoading` method offered by the base Eloquent model class. Typically, you should call this method within the `boot` method of your application's `AppServiceProvider` class.

The `preventLazyLoading` method accepts an optional boolean argument that indicates if lazy loading should be prevented. For example, you may wish to only disable lazy loading in non-production environments so that your production environment will continue to function normally even if a lazy loaded relationship is accidentally present in production code:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

After preventing lazy loading, Eloquent will throw a `Illuminate\Database\LazyLoadingViolationException` exception when your application attempts to lazy load any Eloquent relationship.

You may customize the behavior of lazy loading violations using the `handleLazyLoadingViolationsUsing` method. For example, using this method, you may instruct lazy loading violations to only be logged instead of interrupting the application's execution with exceptions:

```
Model::handleLazyLoadingViolationUsing(function (Model $model, string $relation) {
    $class = $model::class;

    info("Attempted to lazy load [$relation] on model [$class].");
});
```

Inserting and Updating Related Models

The `save` Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to add a new comment to a post. Instead of manually setting the `post_id` attribute on the `Comment` model you may insert the comment using the relationship's `save` method:

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

Note that we did not access the `comments` relationship as a dynamic property. Instead, we called the `comments` method to obtain an instance of the relationship. The `save` method will automatically add the appropriate `post_id` value to the new `Comment` model.

If you need to save multiple related models, you may use the `saveMany` method:

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
```

The `save` and `saveMany` methods will persist the given model instances, but will not add the newly persisted models to any in-memory relationships that are already loaded onto the parent model. If you plan on accessing the relationship after using the `save` or `saveMany` methods, you may wish to use the `refresh` method to reload the model and its relationships:

```
$post->comments()->save($comment);

$post->refresh();

// All comments, including the newly saved comment...
$post->comments;
```

Recursively Saving Models and Relationships

If you would like to `save` your model and all of its associated relationships, you may use the `push` method. In this example, the `Post` model will be saved as well as its comments and the comment's authors:

```
$post = Post::find(1);

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

The `pushQuietly` method may be used to save a model and its associated relationships without raising any events:

```
$post->pushQuietly();
```

The `create` Method

In addition to the `save` and `saveMany` methods, you may also use the `create` method, which accepts an array of attributes, creates a model, and inserts it into the database. The difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`. The newly created model will be returned by the `create` method:

```
use App\Models\Post;

$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

You may use the `createMany` method to create multiple related models:

```
$post = Post::find(1);

$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

The `createQuietly` and `createManyQuietly` methods may be used to create a model(s) without dispatching any events:

```
$user = User::find(1);

$user->posts()->createQuietly([
    'title' => 'Post title.',
]);

$user->posts()->createManyQuietly([
    ['title' => 'First post.'],
    ['title' => 'Second post.'],
]);
```

You may also use the `findOrNew`, `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods to [create and update models on relationships](#).



Before using the `create` method, be sure to review the [mass assignment](#) documentation.

Belongs To Relationships

If you would like to assign a child model to a new parent model, you may use the `associate` method. In this example, the `User` model defines a `belongsTo` relationship to the `Account` model. This `associate` method will set the foreign key on the child model:

```
use App\Models\Account;

$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

To remove a parent model from a child model, you may use the `dissociate` method. This method will set the relationship's foreign key to `null`:

```
$user->account()->dissociate();

$user->save();
```

Many to Many Relationships

Attaching / Detaching

Eloquent also provides methods to make working with many-to-many relationships more convenient. For example, let's imagine a user can have many roles and a role can have many users. You may use the `attach` method to attach a role to a user by inserting a record in the relationship's intermediate table:

```
use App\Models\User;

$user = User::find(1);

$user->roles()->attach($roleId);
```

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Sometimes it may be necessary to remove a role from a user. To remove a many-to-many relationship record, use the `detach` method. The `detach` method will delete the appropriate record out of the intermediate table; however, both models will remain in the database:

```
// Detach a single role from the user...
$user->roles()->detach($roleId);

// Detach all roles from the user...
$user->roles()->detach();
```

For convenience, `attach` and `detach` also accept arrays of IDs as input:

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires],
]);
});
```

Syncing Associations

You may also use the `sync` method to construct many-to-many associations. The `sync` method accepts an array of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the given array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
```

You may also pass additional intermediate table values with the IDs:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

If you would like to insert the same intermediate table values with each of the synced model IDs, you may use the `syncWithPivotValues` method:

```
$user->roles()->syncWithPivotValues([1, 2, 3], ['active' => true]);
```

If you do not want to detach existing IDs that are missing from the given array, you may use the `syncWithoutDetaching` method:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

Toggling Associations

The many-to-many relationship also provides a `toggle` method which "toggles" the attachment status of the given related model IDs. If the given ID is currently attached, it will be detached. Likewise, if it is currently detached, it will be attached:

```
$user->roles()->toggle([1, 2, 3]);
```

You may also pass additional intermediate table values with the IDs:

```
$user->roles()->toggle([
    1 => ['expires' => true],
    2 => ['expires' => true],
]);
});
```

Updating a Record on the Intermediate Table

If you need to update an existing row in your relationship's intermediate table, you may use the `updateExistingPivot` method. This method accepts the intermediate record foreign key and an array of attributes to update:

```
$user = User::find(1);

$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

Touching Parent Timestamps

When a model defines a `belongsTo` or `belongsToMany` relationship to another model, such as a `Comment` which belongs to a `Post`, it is sometimes helpful to update the parent's timestamp when the child model is updated.

For example, when a `Comment` model is updated, you may want to automatically "touch" the `updated_at` timestamp of the owning `Post` so that it is set to the current date and time. To accomplish this, you may add a `touches` property to your child model containing the names of the relationships that should have their `updated_at` timestamps updated when the child model is updated:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```



Parent model timestamps will only be updated if the child model is updated using Eloquent's `save` method.

Eloquent: Collections

- [Introduction](#)
- [Available Methods](#)
- [Custom Collections](#)

Introduction

All Eloquent methods that return more than one model result will return instances of the `Illuminate\Database\Eloquent\Collection` class, including results retrieved via the `get` method or accessed via a relationship. The Eloquent collection object extends Laravel's [base collection](#), so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models. Be sure to review the Laravel collection documentation to learn all about these helpful methods!

All collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, as previously mentioned, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, we may remove all inactive models and then gather the first name for each remaining user:

```
$names = User::all()->reject(function (User $user) {
    return $user->active === false;
})->map(function (User $user) {
    return $user->name;
});
```

Eloquent Collection Conversion

While most Eloquent collection methods return a new instance of an Eloquent collection, the `collapse`, `flatten`, `flip`, `keys`, `pluck`, and `zip` methods return a [base collection](#) instance. Likewise, if a `map` operation returns a collection that does not contain any Eloquent models, it will be converted to a base collection instance.

Available Methods

All Eloquent collections extend the base [Laravel collection](#) object; therefore, they inherit all of the powerful methods provided by the base collection class.

In addition, the `Illuminate\Database\Eloquent\Collection` class provides a superset of methods to aid with managing your model collections. Most methods return `Illuminate\Database\Eloquent\Collection` instances; however, some methods, like `modelKeys`, return an `Illuminate\Support\Collection` instance.

<code>append</code>	<code>fresh</code>	<code>makeVisible</code>
<code>contains</code>	<code>intersect</code>	<code>makeHidden</code>
<code>diff</code>	<code>load</code>	<code>only</code>
<code>except</code>	<code>loadMissing</code>	<code>setVisible</code>
<code>find</code>	<code>modelKeys</code>	<code>setHidden</code>

[toQuery](#)[unique](#)**append(\$attributes)**

The `append` method may be used to indicate that an attribute should be appended for every model in the collection. This method accepts an array of attributes or a single attribute:

```
$users->append('team');

$users->append(['team', 'is_admin']);
```

contains(\$key, \$operator = null, \$value = null)

The `contains` method may be used to determine if a given model instance is contained by the collection. This method accepts a primary key or a model instance:

```
$users->contains(1);

$users->contains(User::find(1));
```

diff(\$items)

The `diff` method returns all of the models that are not present in the given collection:

```
use App\Models\User;

$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());
```

except(\$keys)

The `except` method returns all of the models that do not have the given primary keys:

```
$users = $users->except([1, 2, 3]);
```

find(\$key)

The `find` method returns the model that has a primary key matching the given key. If `$key` is a model instance, `find` will attempt to return a model matching the primary key. If `$key` is an array of keys, `find` will return all models which have a primary key in the given array:

```
$users = User::all();

$user = $users->find(1);
```

fresh(\$with = [])

The `fresh` method retrieves a fresh instance of each model in the collection from the database. In addition, any specified relationships will be eager loaded:

```
$users = $users->fresh();

$users = $users->fresh('comments');
```

intersect(\$items)

The `intersect` method returns all of the models that are also present in the given collection:

```
use App\Models\User;

$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());
```

load(\$relations)

The `load` method eager loads the given relationships for all models in the collection:

```
$users->load(['comments', 'posts']);

$users->load('comments.author');

$users->load(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

loadMissing(\$relations)

The `loadMissing` method eager loads the given relationships for all models in the collection if the relationships are not already loaded:

```
$users->loadMissing(['comments', 'posts']);

$users->loadMissing('comments.author');

$users->loadMissing(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

modelKeys()

The `modelKeys` method returns the primary keys for all models in the collection:

```
$users->modelKeys();

// [1, 2, 3, 4, 5]
```

makeVisible(\$attributes)

The `makeVisible` method makes attributes visible that are typically "hidden" on each model in the collection:

```
$users = $users->makeVisible(['address', 'phone_number']);
```

makeHidden(\$attributes)

The `makeHidden` method hides attributes that are typically "visible" on each model in the collection:

```
$users = $users->makeHidden(['address', 'phone_number']);
```

only(\$keys)

The `only` method returns all of the models that have the given primary keys:

```
$users = $users->only([1, 2, 3]);
```

setVisible(\$attributes)

The `setVisible` method temporarily overrides all of the visible attributes on each model in the collection:

```
$users = $users->setVisible(['id', 'name']);
```

`setHidden($attributes)`

The `setHidden` method temporarily overrides all of the hidden attributes on each model in the collection:

```
$users = $users->setHidden(['email', 'password', 'remember_token']);
```

`toQuery()`

The `toQuery` method returns an Eloquent query builder instance containing a `whereIn` constraint on the collection model's primary keys:

```
use App\Models\User;

$users = User::where('status', 'VIP')->get();

$users->toQuery()->update([
    'status' => 'Administrator',
]);
```

`unique($key = null, $strict = false)`

The `unique` method returns all of the unique models in the collection. Any models of the same type with the same primary key as another model in the collection are removed:

```
$users = $users->unique();
```

Custom Collections

If you would like to use a custom `Collection` object when interacting with a given model, you may define a `newCollection` method on your model:

```
<?php

namespace App\Models;

use App\Support\UserCollection;
use Illuminate\Database\Eloquent\Collection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Create a new Eloquent Collection instance.
     *
     * @param array<int, \Illuminate\Database\Eloquent\Model> $models
     * @return \Illuminate\Database\Eloquent\Collection<int,
     \Illuminate\Database\Eloquent\Model>
     */
    public function newCollection(array $models = []): Collection
    {
        return new UserCollection($models);
    }
}
```

Once you have defined a `newCollection` method, you will receive an instance of your custom collection anytime Eloquent would normally return an `\Illuminate\Database\Eloquent\Collection` instance. If you would like to use a custom collection for every model in your application, you should define the `newCollection` method on a base model class that is extended by all of your application's models.

Eloquent: Mutators & Casting

- [Introduction](#)
- [Accessors and Mutators](#)
 - [Defining an Accessor](#)
 - [Defining a Mutator](#)
- [Attribute Casting](#)
 - [Array and JSON Casting](#)
 - [Date Casting](#)
 - [Enum Casting](#)
 - [Encrypted Casting](#)
 - [Query Time Casting](#)
- [Custom Casts](#)
 - [Value Object Casting](#)
 - [Array / JSON Serialization](#)
 - [Inbound Casting](#)
 - [Cast Parameters](#)
 - [Castables](#)

Introduction

Accessors, mutators, and attribute casting allow you to transform Eloquent attribute values when you retrieve or set them on model instances. For example, you may want to use the [Laravel encrypter](#) to encrypt a value while it is stored in the database, and then automatically decrypt the attribute when you access it on an Eloquent model. Or, you may want to convert a JSON string that is stored in your database to an array when it is accessed via your Eloquent model.

Accessors and Mutators

Defining an Accessor

An accessor transforms an Eloquent attribute value when it is accessed. To define an accessor, create a protected method on your model to represent the accessible attribute. This method name should correspond to the "camel case" representation of the true underlying model attribute / database column when applicable.

In this example, we'll define an accessor for the `first_name` attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the `first_name` attribute. All attribute accessor / mutator methods must declare a return type-hint of `Illuminate\Database\Eloquent\Castable`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the user's first name.
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
        );
    }
}
```

All accessor methods return an `Attribute` instance which defines how the attribute will be accessed and, optionally, mutated. In this example, we are only defining how the attribute will be accessed. To do so, we supply the `get` argument to the `Attribute` class constructor.

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may simply access the `first_name` attribute on a model instance:

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```



If you would like these computed values to be added to the array / JSON representations of your model, you will need to append them.

Building Value Objects From Multiple Attributes

Sometimes your accessor may need to transform multiple model attributes into a single "value object". To do so, your `get` closure may accept a second argument of `$attributes`, which will be automatically supplied to the closure and will contain an array of all of the model's current attributes:

```

use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * Interact with the user's address.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
    );
}

```

Accessor Caching

When returning value objects from accessors, any changes made to the value object will automatically be synced back to the model before the model is saved. This is possible because Eloquent retains instances returned by accessors so it can return the same instance each time the accessor is invoked:

```

use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Line 1 Value';
$user->address->lineTwo = 'Updated Address Line 2 Value';

$user->save();

```

However, you may sometimes wish to enable caching for primitive values like strings and booleans, particularly if they are computationally intensive. To accomplish this, you may invoke the `shouldCache` method when defining your accessor:

```

protected function hash(): Attribute
{
    return Attribute::make(
        get: fn (string $value) => bcrypt(gzuncompress($value)),
        ->shouldCache(),
    );
}

```

If you would like to disable the object caching behavior of attributes, you may invoke the `withoutObjectCaching` method when defining the attribute:

```
/**
 * Interact with the user's address.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
    )->withoutObjectCaching();
}
```

Defining a Mutator

A mutator transforms an Eloquent attribute value when it is set. To define a mutator, you may provide the `set` argument when defining your attribute. Let's define a mutator for the `first_name` attribute. This mutator will be automatically called when we attempt to set the value of the `first_name` attribute on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Interact with the user's first name.
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
            set: fn (string $value) => strtolower($value),
        );
    }
}
```

The mutator closure will receive the value that is being set on the attribute, allowing you to manipulate the value and return the manipulated value. To use our mutator, we only need to set the `first_name` attribute on an Eloquent model:

```
use App\Models\User;

$user = User::find(1);

$user->first_name = 'Sally';
```

In this example, the `set` callback will be called with the value `Sally`. The mutator will then apply the `strtolower` function to the name and set its resulting value in the model's internal `$attributes` array.

Mutating Multiple Attributes

Sometimes your mutator may need to set multiple attributes on the underlying model. To do so, you may return an array from the `set` closure. Each key in the array should correspond with an underlying attribute / database column associated with the model:

```
use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * Interact with the user's address.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
        set: fn (Address $value) => [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ],
    );
}
```

Attribute Casting

Attribute casting provides functionality similar to accessors and mutators without requiring you to define any additional methods on your model. Instead, your model's `$casts` property provides a convenient method of converting attributes to common data types.

The `$casts` property should be an array where the key is the name of the attribute being cast and the value is the type you wish to cast the column to. The supported cast types are:

- `array`
- `AsStringable::class`
- `boolean`
- `collection`
- `date`
- `datetime`
- `immutable_date`
- `immutable_datetime`
- `decimal:<precision>`
- `double`
- `encrypted`
- `encrypted:array`
- `encrypted:collection`
- `encrypted:object`
- `float`
- `hashed`
- `integer`
- `object`
- `real`
- `string`
- `timestamp`

To demonstrate attribute casting, let's cast the `is_admin` attribute, which is stored in our database as an integer (`0` or `1`) to a boolean value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

After defining the cast, the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer:

```
$user = App\Models\User::find(1);

if ($user->is_admin) {
    // ...
}
```

If you need to add a new, temporary cast at runtime, you may use the `mergeCasts` method. These cast definitions will be added to any of the casts already defined on the model:

```
$user->mergeCasts([
    'is_admin' => 'integer',
    'options' => 'object',
]);
```



Attributes that are `null` will not be cast. In addition, you should never define a cast (or an attribute) that has the same name as a relationship or assign a cast to the model's primary key.

Stringable Casting

You may use the `Illuminate\Database\Eloquent\Cast\AsStringable` cast class to cast a model attribute to a fluent `Illuminate\Support\Stringable` object:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\CastableAsStringable;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'directory' => AsStringable::class,
    ];
}
```

Array and JSON Casting

The `array` cast is particularly useful when working with columns that are stored as serialized JSON. For example, if your database has a `JSON` or `TEXT` field type that contains serialized JSON, adding the `array` cast to that attribute will automatically deserialize the attribute to a PHP array when you access it on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

Once the cast is defined, you may access the `options` attribute and it will automatically be deserialized from JSON into a PHP array. When you set the value of the `options` attribute, the given array will automatically be serialized back into JSON for storage:

```
use App\Models\User;

$user = User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

To update a single field of a JSON attribute with a more terse syntax, you may [make the attribute mass assignable](#) and use the `->` operator when calling the `update` method:

```
$user = User::find(1);

$user->update(['options->key' => 'value']);
```

Array Object and Collection Casting

Although the standard `array` cast is sufficient for many applications, it does have some disadvantages. Since the `array` cast returns a primitive type, it is not possible to mutate an offset of the array directly. For example, the following code will trigger a PHP error:

```
$user = User::find(1);

$user->options['key'] = $value;
```

To solve this, Laravel offers an `AsArrayObject` cast that casts your JSON attribute to an [ArrayObject](#) class. This feature is implemented using Laravel's [custom cast](#) implementation, which allows Laravel to intelligently cache and transform the mutated object such that individual offsets may be modified without triggering a PHP error. To use the `AsArrayObject` cast, simply assign it to an attribute:

```
use Illuminate\Database\Eloquent\Casters\AsArrayObject;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'options' => AsArrayObject::class,
];
```

Similarly, Laravel offers an `AsCollection` cast that casts your JSON attribute to a Laravel [Collection](#) instance:

```
use Illuminate\Database\Eloquent\Cast\AsCollection;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'options' => AsCollection::class,
];
```

If you would like the `AsCollection` cast to instantiate a custom collection class instead of Laravel's base collection class, you may provide the collection class name as a cast argument:

```
use App\Collections\OptionCollection;
use Illuminate\Database\Eloquent\Cast\AsCollection;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'options' => AsCollection::class.':'.OptionCollection::class,
];
```

Date Casting

By default, Eloquent will cast the `created_at` and `updated_at` columns to instances of `Carbon`, which extends the PHP `DateTime` class and provides an assortment of helpful methods. You may cast additional date attributes by defining additional date casts within your model's `$casts` property array. Typically, dates should be cast using the `datetime` or `immutable_datetime` cast types.

When defining a `date` or `datetime` cast, you may also specify the date's format. This format will be used when the [model is serialized to an array or JSON](#):

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'created_at' => 'datetime:Y-m-d',
];
```

When a column is cast as a date, you may set the corresponding model attribute value to a UNIX timestamp, date string (`Y-m-d`), date-time string, or a `DateTime` / `Carbon` instance. The date's value will be correctly converted and stored in your database.

You may customize the default serialization format for all of your model's dates by defining a `serializeDate` method on your model. This method does not affect how your dates are formatted for storage in the database:

```
/**
 * Prepare a date for array / JSON serialization.
 */
protected function serializeDate(DateTimeInterface $date): string
{
    return $date->format('Y-m-d');
}
```

To specify the format that should be used when actually storing a model's dates within your database, you should define a `$dateFormat` property on your model:

```
/*
 * The storage format of the model's date columns.
 *
 * @var string
 */
protected $dateFormat = 'U';
```

Date Casting, Serialization, and Timezones

By default, the `date` and `datetime` casts will serialize dates to a UTC ISO-8601 date string (`YYYY-MM-DDTHH:MM:SS.uuuuuuZ`), regardless of the timezone specified in your application's `timezone` configuration option. You are strongly encouraged to always use this serialization format, as well as to store your application's dates in the UTC timezone by not changing your application's `timezone` configuration option from its default `UTC` value. Consistently using the UTC timezone throughout your application will provide the maximum level of interoperability with other date manipulation libraries written in PHP and JavaScript.

If a custom format is applied to the `date` or `datetime` cast, such as `datetime:Y-m-d H:i:s`, the inner timezone of the Carbon instance will be used during date serialization. Typically, this will be the timezone specified in your application's `timezone` configuration option.

Enum Casting

Eloquent also allows you to cast your attribute values to PHP [Enums](#). To accomplish this, you may specify the attribute and enum you wish to cast in your model's `$casts` property array:

```
use App\Enums\ServerStatus;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'status' => ServerStatus::class,
];
```

Once you have defined the cast on your model, the specified attribute will be automatically cast to and from an enum when you interact with the attribute:

```

if ($server->status == ServerStatus::Provisioned) {
    $server->status = ServerStatus::Ready;

    $server->save();
}

```

Casting Arrays of Enums

Sometimes you may need your model to store an array of enum values within a single column. To accomplish this, you may utilize the `AsEnumArrayObject` or `AsEnumCollection` casts provided by Laravel:

```

use App\Enums\ServerStatus;
use Illuminate\Database\Eloquent\Casts\AsEnumCollection;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'statuses' => AsEnumCollection::class . ':' . ServerStatus::class,
];

```

Encrypted Casting

The `encrypted` cast will encrypt a model's attribute value using Laravel's built-in [encryption](#) features. In addition, the `encrypted:array`, `encrypted:collection`, `encrypted:object`, `AsEncryptedArrayObject`, and `AsEncryptedCollection` casts work like their unencrypted counterparts; however, as you might expect, the underlying value is encrypted when stored in your database.

As the final length of the encrypted text is not predictable and is longer than its plain text counterpart, make sure the associated database column is of `TEXT` type or larger. In addition, since the values are encrypted in the database, you will not be able to query or search encrypted attribute values.

Key Rotation

As you may know, Laravel encrypts strings using the `key` configuration value specified in your application's `app` configuration file. Typically, this value corresponds to the value of the `APP_KEY` environment variable. If you need to rotate your application's encryption key, you will need to manually re-encrypt your encrypted attributes using the new key.

Query Time Casting

Sometimes you may need to apply casts while executing a query, such as when selecting a raw value from a table. For example, consider the following query:

```

use App\Models\Post;
use App\Models\User;

$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
        ->whereColumn('user_id', 'users.id')
])->get();

```

The `last_posted_at` attribute on the results of this query will be a simple string. It would be wonderful if we could apply a `datetime` cast to this attribute when executing the query. Thankfully, we may accomplish this using the `withCasts`

method:

```
$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
        ->whereColumn('user_id', 'users.id')
])->withCasts([
    'last_posted_at' => 'datetime'
])->get();
```

Custom Casts

Laravel has a variety of built-in, helpful cast types; however, you may occasionally need to define your own cast types. To create a cast, execute the `make:cast` Artisan command. The new cast class will be placed in your `app/Casts` directory:

```
php artisan make:cast Json
```

All custom cast classes implement the `CastsAttributes` interface. Classes that implement this interface must define a `get` and `set` method. The `get` method is responsible for transforming a raw value from the database into a cast value, while the `set` method should transform a cast value into a raw value that can be stored in the database. As an example, we will re-implement the built-in `json` cast type as a custom cast type:

```
<?php

namespace App\Cast;

use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use Illuminate\Database\Eloquent\Model;

class Json implements CastsAttributes
{
    /**
     * Cast the given value.
     *
     * @param array<string, mixed> $attributes
     * @return array<string, mixed>
     */
    public function get(Model $model, string $key, mixed $value, array $attributes): array
    {
        return json_decode($value, true);
    }

    /**
     * Prepare the given value for storage.
     *
     * @param array<string, mixed> $attributes
     */
    public function set(Model $model, string $key, mixed $value, array $attributes): string
    {
        return json_encode($value);
    }
}
```

Once you have defined a custom cast type, you may attach it to a model attribute using its class name:

```
<?php

namespace App\Models;

use App\Cast\Json;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => Json::class,
    ];
}
```

Value Object Casting

You are not limited to casting values to primitive types. You may also cast values to objects. Defining custom casts that cast values to objects is very similar to casting to primitive types; however, the `set` method should return an array of key / value pairs that will be used to set raw, storables values on the model.

As an example, we will define a custom cast class that casts multiple model values into a single `Address` value object. We will assume the `Address` value has two public properties: `lineOne` and `lineTwo`:

```

<?php

namespace App\Cast;

use App\ValueObjects\Address as AddressValueObject;
use Illuminate\Contracts\Database\Eloquent\CastAttributes;
use Illuminate\Database\Eloquent\Model;
use InvalidArgumentException;

class Address implements CastAttributes
{
    /**
     * Cast the given value.
     *
     * @param array<string, mixed> $attributes
     */
    public function get(Model $model, string $key, mixed $value, array $attributes): AddressValueObject
    {
        return new AddressValueObject(
            $attributes['address_line_one'],
            $attributes['address_line_two']
        );
    }

    /**
     * Prepare the given value for storage.
     *
     * @param array<string, mixed> $attributes
     * @return array<string, string>
     */
    public function set(Model $model, string $key, mixed $value, array $attributes): array
    {
        if (! $value instanceof AddressValueObject) {
            throw new InvalidArgumentException('The given value is not an Address instance.');
        }

        return [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ];
    }
}

```

When casting to value objects, any changes made to the value object will automatically be synced back to the model before the model is saved:

```

use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Value';

$user->save();

```



If you plan to serialize your Eloquent models containing value objects to JSON or arrays, you should implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces on the value object.

Value Object Caching

When attributes that are cast to value objects are resolved, they are cached by Eloquent. Therefore, the same object instance will be returned if the attribute is accessed again.

If you would like to disable the object caching behavior of custom cast classes, you may declare a public `withoutObjectCaching` property on your custom cast class:

```
class Address implements CastsAttributes
{
    public bool $withoutObjectCaching = true;

    // ...
}
```

Array / JSON Serialization

When an Eloquent model is converted to an array or JSON using the `toArray` and `toJson` methods, your custom cast value objects will typically be serialized as well as long as they implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces. However, when using value objects provided by third-party libraries, you may not have the ability to add these interfaces to the object.

Therefore, you may specify that your custom cast class will be responsible for serializing the value object. To do so, your custom cast class should implement the

`Illuminate\Contracts\Database\Eloquent\SerializesCastableAttributes` interface. This interface states that your class should contain a `serialize` method which should return the serialized form of your value object:

```
/**
 * Get the serialized representation of the value.
 *
 * @param array<string, mixed> $attributes
 */
public function serialize(Model $model, string $key, mixed $value, array $attributes): string
{
    return (string) $value;
}
```

Inbound Casting

Occasionally, you may need to write a custom cast class that only transforms values that are being set on the model and does not perform any operations when attributes are being retrieved from the model.

Inbound only custom casts should implement the `CastsInboundAttributes` interface, which only requires a `set` method to be defined. The `make:cast` Artisan command may be invoked with the `--inbound` option to generate an inbound only cast class:

```
php artisan make:cast Hash --inbound
```

A classic example of an inbound only cast is a "hashing" cast. For example, we may define a cast that hashes inbound values via a given algorithm:

```
<?php

namespace App\Cast;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Database\Eloquent\Model;

class Hash implements Castable
{
    /**
     * Create a new cast class instance.
     */
    public function __construct(
        protected string|null $algorithm = null,
    ) {}

    /**
     * Prepare the given value for storage.
     *
     * @param array<string, mixed> $attributes
     */
    public function set(Model $model, string $key, mixed $value, array $attributes): string
    {
        return is_null($this->algorithm)
            ? bcrypt($value)
            : hash($this->algorithm, $value);
    }
}
```

Cast Parameters

When attaching a custom cast to a model, cast parameters may be specified by separating them from the class name using a `:` character and comma-delimiting multiple parameters. The parameters will be passed to the constructor of the cast class:

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'secret' => Hash::class.':sha256',
];
```

Castables

You may want to allow your application's value objects to define their own custom cast classes. Instead of attaching the custom cast class to your model, you may alternatively attach a value object class that implements the `Illuminate\Contracts\Database\Eloquent\Castable` interface:

```
use App\ValueObjects\Address;

protected $casts = [
    'address' => Address::class,
];
```

Objects that implement the `Castable` interface must define a `castUsing` method that returns the class name of the custom caster class that is responsible for casting to and from the `Castable` class:

```
<?php

namespace App\ValueObjects;

use Illuminate\Contracts\Database\Eloquent\Castable;
use App\Cast\Address as AddressCast;

class Address implements Castable
{
    /**
     * Get the name of the caster class to use when casting from / to this cast target.
     *
     * @param array<string, mixed> $arguments
     */
    public static function castUsing(array $arguments): string
    {
        return AddressCast::class;
    }
}
```

When using `Castable` classes, you may still provide arguments in the `$casts` definition. The arguments will be passed to the `castUsing` method:

```
use App\ValueObjects\Address;

protected $casts = [
    'address' => Address::class.':argument',
];
```

Castables & Anonymous Cast Classes

By combining "castables" with PHP's [anonymous classes](#), you may define a value object and its casting logic as a single castable object. To accomplish this, return an anonymous class from your value object's `castUsing` method. The anonymous class should implement the `CastsAttributes` interface:

```
<?php

namespace App\ValueObjects;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Contracts\Database\Eloquent\CastableAttributes;

class Address implements Castable
{
    // ...

    /**
     * Get the caster class to use when casting from / to this cast target.
     *
     * @param array<string, mixed> $arguments
     */
    public static function castUsing(array $arguments): CastableAttributes
    {
        return new class implements CastableAttributes
        {
            public function get(Model $model, string $key, mixed $value, array $attributes): Address
            {
                return new Address(
                    $attributes['address_line_one'],
                    $attributes['address_line_two']
                );
            }

            public function set(Model $model, string $key, mixed $value, array $attributes): array
            {
                return [
                    'address_line_one' => $value->lineOne,
                    'address_line_two' => $value->lineTwo,
                ];
            }
        };
    }
}
```

Eloquent: API Resources

- [Introduction](#)
- [Generating Resources](#)
- [Concept Overview](#)
 - [Resource Collections](#)
- [Writing Resources](#)
 - [Data Wrapping](#)
 - [Pagination](#)
 - [Conditional Attributes](#)
 - [Conditional Relationships](#)
 - [Adding Meta Data](#)
- [Resource Responses](#)

Introduction

When building an API, you may need a transformation layer that sits between your Eloquent models and the JSON responses that are actually returned to your application's users. For example, you may wish to display certain attributes for a subset of users and not others, or you may wish to always include certain relationships in the JSON representation of your models. Eloquent's resource classes allow you to expressively and easily transform your models and model collections into JSON.

Of course, you may always convert Eloquent models or collections to JSON using their `toJson` methods; however, Eloquent resources provide more granular and robust control over the JSON serialization of your models and their relationships.

Generating Resources

To generate a resource class, you may use the `make:resource` Artisan command. By default, resources will be placed in the `app/Http/Resources` directory of your application. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource UserResource
```

Resource Collections

In addition to generating resources that transform individual models, you may generate resources that are responsible for transforming collections of models. This allows your JSON responses to include links and other meta information that is relevant to an entire collection of a given resource.

To create a resource collection, you should use the `--collection` flag when creating the resource. Or, including the word `Collection` in the resource name will indicate to Laravel that it should create a collection resource. Collection resources extend the `Illuminate\Http\Resources\Json\ResourceCollection` class:

```
php artisan make:resource User --collection
php artisan make:resource UserCollection
```

Concept Overview



This is a high-level overview of resources and resource collections. You are highly encouraged to read the other sections of this documentation to gain a deeper understanding of the customization and power offered to you by resources.

Before diving into all of the options available to you when writing resources, let's first take a high-level look at how resources are used within Laravel. A resource class represents a single model that needs to be transformed into a JSON structure. For example, here is a simple `UserResource` resource class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Every resource class defines a `toArray` method which returns the array of attributes that should be converted to JSON when the resource is returned as a response from a route or controller method.

Note that we can access model properties directly from the `$this` variable. This is because a resource class will automatically proxy property and method access down to the underlying model for convenient access. Once the resource is defined, it may be returned from a route or controller. The resource accepts the underlying model instance via its constructor:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});
```

Resource Collections

If you are returning a collection of resources or a paginated response, you should use the `collection` method provided by your resource class when creating the resource instance in your route or controller:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Note that this does not allow any addition of custom meta data that may need to be returned with your collection. If you would like to customize the resource collection response, you may create a dedicated resource to represent the collection:

```
php artisan make:resource UserCollection
```

Once the resource collection class has been generated, you may easily define any meta data that should be included with the response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @return array<int|string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

After defining your resource collection, it may be returned from a route or controller:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Preserving Collection Keys

When returning a resource collection from a route, Laravel resets the collection's keys so that they are in numerical order. However, you may add a `preserveKeys` property to your resource class indicating whether a collection's original keys should be preserved:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

When the `preserveKeys` property is set to `true`, collection keys will be preserved when the collection is returned from a route or controller:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all()->keyBy('id'));
});
```

Customizing the Underlying Resource Class

Typically, the `$this->collection` property of a resource collection is automatically populated with the result of mapping each item of the collection to its singular resource class. The singular resource class is assumed to be the collection's class name without the trailing `Collection` portion of the class name. In addition, depending on your personal preference, the singular resource class may or may not be suffixed with `Resource`.

For example, `UserCollection` will attempt to map the given user instances into the `UserResource` resource. To customize this behavior, you may override the `$collects` property of your resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * The resource that this resource collects.
     *
     * @var string
     */
    public $collects = Member::class;
}
```

Writing Resources



If you have not read the [concept overview](#), you are highly encouraged to do so before proceeding with this documentation.

Resources only need to transform a given model into an array. So, each resource contains a `toArray` method which translates your model's attributes into an API friendly array that can be returned from your application's routes or controllers:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Once a resource has been defined, it may be returned directly from a route or controller:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});
```

Relationships

If you would like to include related resources in your response, you may add them to the array returned by your resource's `toArray` method. In this example, we will use the `PostResource` resource's `collection` method to add the user's blog posts to the resource response:

```
use App\Http\Resources\PostResource;
use Illuminate\Http\Request;

/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```



If you would like to include relationships only when they have already been loaded, check out the documentation on [conditional relationships](#).

Resource Collections

While resources transform a single model into an array, resource collections transform a collection of models into an array. However, it is not absolutely necessary to define a resource collection class for each one of your models since all resources provide a `collection` method to generate an "ad-hoc" resource collection on the fly:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

However, if you need to customize the meta data returned with the collection, it is necessary to define your own resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Like singular resources, resource collections may be returned directly from routes or controllers:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Data Wrapping

By default, your outermost resource is wrapped in a `data` key when the resource response is converted to JSON. So, for example, a typical resource collection response looks like the following:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ]
}
```

If you would like to disable the wrapping of the outermost resource, you should invoke the `withoutWrapping` method on the base `Illuminate\Http\Resources\Json\JsonResource` class. Typically, you should call this method from your `AppServiceProvider` or another service provider that is loaded on every request to your application:

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        JsonResource::withoutWrapping();
    }
}
```



The `withoutWrapping` method only affects the outermost response and will not remove `data` keys that you manually add to your own resource collections.

Wrapping Nested Resources

You have total freedom to determine how your resource's relationships are wrapped. If you would like all resource collections to be wrapped in a `data` key, regardless of their nesting, you should define a resource collection class for

each resource and return the collection within a `data` key.

You may be wondering if this will cause your outermost resource to be wrapped in two `data` keys. Don't worry, Laravel will never let your resources be accidentally double-wrapped, so you don't have to be concerned about the nesting level of the resource collection you are transforming:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return ['data' => $this->collection];
    }
}
```

Data Wrapping and Pagination

When returning paginated collections via a resource response, Laravel will wrap your resource data in a `data` key even if the `withoutWrapping` method has been called. This is because paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links":{
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/users",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Pagination

You may pass a Laravel paginator instance to the `collection` method of a resource or to a custom resource collection:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links":{
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/users",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Customizing the Pagination Information

If you would like to customize the information included in the `links` or `meta` keys of the pagination response, you may define a `paginationInformation` method on the resource. This method will receive the `$paginated` data and the array of `$default` information, which is an array containing the `links` and `meta` keys:

```
/**
 * Customize the pagination information for the resource.
 *
 * @param \Illuminate\Http\Request $request
 * @param array $paginated
 * @param array $default
 * @return array
 */
public function paginationInformation($request, $paginated, $default)
{
    $default['links']['custom'] = 'https://example.com';

    return $default;
}
```

Conditional Attributes

Sometimes you may wish to only include an attribute in a resource response if a given condition is met. For example, you may wish to only include a value if the current user is an "administrator". Laravel provides a variety of helper methods to

assist you in this situation. The `when` method may be used to conditionally add an attribute to a resource response:

```
/*
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when($request->user()->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, the `secret` key will only be returned in the final resource response if the authenticated user's `isAdmin` method returns `true`. If the method returns `false`, the `secret` key will be removed from the resource response before it is sent to the client. The `when` method allows you to expressively define your resources without resorting to conditional statements when building the array.

The `when` method also accepts a closure as its second argument, allowing you to calculate the resulting value only if the given condition is `true`:

```
'secret' => $this->when($request->user()->isAdmin(), function () {
    return 'secret-value';
}),
```

The `whenHas` method may be used to include an attribute if it is actually present on the underlying model:

```
'name' => $this->whenHas('name'),
```

Additionally, the `whenNotNull` method may be used to include an attribute in the resource response if the attribute is not null:

```
'name' => $this->whenNotNull($this->name),
```

Merging Conditional Attributes

Sometimes you may have several attributes that should only be included in the resource response based on the same condition. In this case, you may use the `mergeWhen` method to include the attributes in the response only when the given condition is `true`:

```
/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen($request->user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

Again, if the given condition is `false`, these attributes will be removed from the resource response before it is sent to the client.



The `mergeWhen` method should not be used within arrays that mix string and numeric keys. Furthermore, it should not be used within arrays with numeric keys that are not ordered sequentially.

Conditional Relationships

In addition to conditionally loading attributes, you may conditionally include relationships on your resource responses based on if the relationship has already been loaded on the model. This allows your controller to decide which relationships should be loaded on the model and your resource can easily include them only when they have actually been loaded. Ultimately, this makes it easier to avoid "N+1" query problems within your resources.

The `whenLoaded` method may be used to conditionally load a relationship. In order to avoid unnecessarily loading relationships, this method accepts the name of the relationship instead of the relationship itself:

```
use App\Http\Resources\PostResource;

/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, if the relationship has not been loaded, the `posts` key will be removed from the resource response before it is sent to the client.

Conditional Relationship Counts

In addition to conditionally including relationships, you may conditionally include relationship "counts" on your resource responses based on if the relationship's count has been loaded on the model:

```
new UserResource($user->loadCount('posts'));
```

The `whenCounted` method may be used to conditionally include a relationship's count in your resource response. This method avoids unnecessarily including the attribute if the relationships' count is not present:

```
/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts_count' => $this->whenCounted('posts'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, if the `posts` relationship's count has not been loaded, the `posts_count` key will be removed from the resource response before it is sent to the client.

Other types of aggregates, such as `avg`, `sum`, `min`, and `max` may also be conditionally loaded using the `whenAggregated` method:

```
'words_avg' => $this->whenAggregated('posts', 'words', 'avg'),
'words_sum' => $this->whenAggregated('posts', 'words', 'sum'),
'words_min' => $this->whenAggregated('posts', 'words', 'min'),
'words_max' => $this->whenAggregated('posts', 'words', 'max'),
```

Conditional Pivot Information

In addition to conditionally including relationship information in your resource responses, you may conditionally include data from the intermediate tables of many-to-many relationships using the `whenPivotLoaded` method. The `whenPivotLoaded` method accepts the name of the pivot table as its first argument. The second argument should be a closure that returns the value to be returned if the pivot information is available on the model:

```
/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_user', function () {
            return $this->pivot->expires_at;
        }),
    ];
}
```

If your relationship is using a [custom intermediate table model](#), you may pass an instance of the intermediate table model as the first argument to the `whenPivotLoaded` method:

```
'expires_at' => $this->whenPivotLoaded(new Membership, function () {
    return $this->pivot->expires_at;
}),
```

If your intermediate table is using an accessor other than `pivot`, you may use the `whenPivotLoadedAs` method:

```
/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function () {
            return $this->subscription->expires_at;
        }),
    ];
}
```

Adding Meta Data

Some JSON API standards require the addition of meta data to your resource and resource collections responses. This often includes things like `links` to the resource or related resources, or meta data about the resource itself. If you need to return additional meta data about a resource, include it in your `toArray` method. For example, you might include `link` information when transforming a resource collection:

```
/*
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

When returning additional meta data from your resources, you never have to worry about accidentally overriding the `links` or `meta` keys that are automatically added by Laravel when returning paginated responses. Any additional `links` you define will be merged with the links provided by the paginator.

Top Level Meta Data

Sometimes you may wish to only include certain meta data with a resource response if the resource is the outermost resource being returned. Typically, this includes meta information about the response as a whole. To define this meta data, add a `with` method to your resource class. This method should return an array of meta data to be included with the resource response only when the resource is the outermost resource being transformed:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource array.
     *
     * @return array<string, mixed>
     */
    public function with(Request $request): array
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}
```

Adding Meta Data When Constructing Resources

You may also add top-level data when constructing resource instances in your route or controller. The `additional` method, which is available on all resources, accepts an array of data that should be added to the resource response:

```
return (new UserCollection(User::all()->load('roles'))->additional(['meta' => [
    'key' => 'value',
]]));
```

Resource Responses

As you have already read, resources may be returned directly from routes and controllers:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});
```

However, sometimes you may need to customize the outgoing HTTP response before it is sent to the client. There are two ways to accomplish this. First, you may chain the `response` method onto the resource. This method will return an `Illuminate\Http\JsonResponse` instance, giving you full control over the response's headers:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
        ->response()
        ->header('X-Value', 'True');
});
```

Alternatively, you may define a `withResponse` method within the resource itself. This method will be called when the resource is returned as the outermost resource in a response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     */
    public function withResponse(Request $request, JsonResponse $response): void
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent: Serialization

- [Introduction](#)
- [Serializing Models and Collections](#)
 - [Serializing to Arrays](#)
 - [Serializing to JSON](#)
- [Hiding Attributes From JSON](#)
- [Appending Values to JSON](#)
- [Date Serialization](#)

Introduction

When building APIs using Laravel, you will often need to convert your models and relationships to arrays or JSON. Eloquent includes convenient methods for making these conversions, as well as controlling which attributes are included in the serialized representation of your models.



For an even more robust way of handling Eloquent model and collection JSON serialization, check out the documentation on [Eloquent API resources](#).

Serializing Models and Collections

Serializing to Arrays

To convert a model and its loaded [relationships](#) to an array, you should use the `toArray` method. This method is recursive, so all attributes and all relations (including the relations of relations) will be converted to arrays:

```
use App\Models\User;

$user = User::with('roles')->first();

return $user->toArray();
```

The `attributesToArray` method may be used to convert a model's attributes to an array but not its relationships:

```
$user = User::first();

return $user->attributesToArray();
```

You may also convert entire [collections](#) of models to arrays by calling the `toArray` method on the collection instance:

```
$users = User::all();

return $users->toArray();
```

Serializing to JSON

To convert a model to JSON, you should use the `toJson` method. Like `toArray`, the `toJson` method is recursive, so all attributes and relations will be converted to JSON. You may also specify any JSON encoding options that are [supported by PHP](#):

```
use App\Models\User;

$user = User::find(1);

return $user->toJson();

return $user->toJson(JSON_PRETTY_PRINT);
```

Alternatively, you may cast a model or collection to a string, which will automatically call the `toJson` method on the model or collection:

```
return (string) User::find(1);
```

Since models and collections are converted to JSON when cast to a string, you can return Eloquent objects directly from your application's routes or controllers. Laravel will automatically serialize your Eloquent models and collections to JSON when they are returned from routes or controllers:

```
Route::get('users', function () {
    return User::all();
});
```

Relationships

When an Eloquent model is converted to JSON, its loaded relationships will automatically be included as attributes on the JSON object. Also, though Eloquent relationship methods are defined using "camel case" method names, a relationship's JSON attribute will be "snake case".

Hiding Attributes From JSON

Sometimes you may wish to limit the attributes, such as passwords, that are included in your model's array or JSON representation. To do so, add a `$hidden` property to your model. Attributes that are listed in the `$hidden` property's array will not be included in the serialized representation of your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```



To hide relationships, add the relationship's method name to your Eloquent model's `$hidden` property.

Alternatively, you may use the `visible` property to define an "allow list" of attributes that should be included in your model's array and JSON representation. All attributes that are not present in the `[$visible]` array will be hidden when the model is converted to an array or JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be visible in arrays.
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

Temporarily Modifying Attribute Visibility

If you would like to make some typically hidden attributes visible on a given model instance, you may use the `makeVisible` method. The `makeVisible` method returns the model instance:

```
return $user->makeVisible('attribute')->toArray();
```

Likewise, if you would like to hide some attributes that are typically visible, you may use the `makeHidden` method:

```
return $user->makeHidden('attribute')->toArray();
```

If you wish to temporarily override all of the visible or hidden attributes, you may use the `setVisible` and `setHidden` methods respectively:

```
return $user->setVisible(['id', 'name'])->toArray();

return $user->setHidden(['email', 'password', 'remember_token'])->toArray();
```

Appending Values to JSON

Occasionally, when converting models to arrays or JSON, you may wish to add attributes that do not have a corresponding column in your database. To do so, first define an [accessor](#) for the value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Determine if the user is an administrator.
     */
    protected function isAdmin(): Attribute
    {
        return new Attribute(
            get: fn () => 'yes',
        );
    }
}
```

If you would like the accessor to always be appended to your model's array and JSON representations, you may add the attribute name to the `appends` property of your model. Note that attribute names are typically referenced using their "snake case" serialized representation, even though the accessor's PHP method is defined using "camel case":

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The accessors to append to the model's array form.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

Once the attribute has been added to the `appends` list, it will be included in both the model's array and JSON representations. Attributes in the `appends` array will also respect the `visible` and `hidden` settings configured on the model.

Appending at Run Time

At runtime, you may instruct a model instance to append additional attributes using the `append` method. Or, you may use the `setAppends` method to override the entire array of appended properties for a given model instance:

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

Date Serialization

Customizing the Default Date Format

You may customize the default serialization format by overriding the `serializeDate` method. This method does not affect how your dates are formatted for storage in the database:

```
/**  
 * Prepare a date for array / JSON serialization.  
 */  
protected function serializeDate(DateTimeInterface $date): string  
{  
    return $date->format('Y-m-d');  
}
```

Customizing the Date Format per Attribute

You may customize the serialization format of individual Eloquent date attributes by specifying the date format in the model's [cast declarations](#):

```
protected $casts = [  
    'birthday' => 'date:Y-m-d',  
    'joined_at' => 'datetime:Y-m-d H:00',  
];
```

Eloquent: Factories

- [Introduction](#)
- [Defining Model Factories](#)
 - [Generating Factories](#)
 - [Factory States](#)
 - [Factory Callbacks](#)
- [Creating Models Using Factories](#)
 - [Instantiating Models](#)
 - [Persisting Models](#)
 - [Sequences](#)
- [Factory Relationships](#)
 - [Has Many Relationships](#)
 - [Belongs To Relationships](#)
 - [Many to Many Relationships](#)
 - [Polymorphic Relationships](#)
 - [Defining Relationships Within Factories](#)
 - [Recycling an Existing Model for Relationships](#)

Introduction

When testing your application or seeding your database, you may need to insert a few records into your database. Instead of manually specifying the value of each column, Laravel allows you to define a set of default attributes for each of your [Eloquent models](#) using model factories.

To see an example of how to write a factory, take a look at the `database/factories/UserFactory.php` file in your application. This factory is included with all new Laravel applications and contains the following factory definition:

```
namespace Database\Factories;

use Illuminate\Support\Str;
use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => '$2y$10$92IXUNpkj00rQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', //
password
            'remember_token' => Str::random(10),
        ];
    }
}
```

As you can see, in their most basic form, factories are classes that extend Laravel's base factory class and define a `definition` method. The `definition` method returns the default set of attribute values that should be applied when creating a model using the factory.

Via the `fake` helper, factories have access to the [Faker](#) PHP library, which allows you to conveniently generate various kinds of random data for testing and seeding.



You can set your application's Faker locale by adding a `faker_locale` option to your `config/app.php` configuration file.

Defining Model Factories

Generating Factories

To create a factory, execute the `make:factory` Artisan command:

```
php artisan make:factory PostFactory
```

The new factory class will be placed in your `database/factories` directory.

Model and Factory Discovery Conventions

Once you have defined your factories, you may use the static `factory` method provided to your models by the `Illuminate\Database\Eloquent\Factories\HasFactory` trait in order to instantiate a factory instance for that model.

The `HasFactory` trait's `factory` method will use conventions to determine the proper factory for the model the trait is assigned to. Specifically, the method will look for a factory in the `Database\Factories` namespace that has a class name matching the model name and is suffixed with `Factory`. If these conventions do not apply to your particular application or factory, you may overwrite the `newFactory` method on your model to return an instance of the model's corresponding factory directly:

```
use Illuminate\Database\Eloquent\Factories\Factory;
use Database\Factories\Administration\FlightFactory;

/**
 * Create a new factory instance for the model.
 */
protected static function newFactory(): Factory
{
    return FlightFactory::new();
}
```

Then, define a `model` property on the corresponding factory:

```
use App\Administration\Flight;
use Illuminate\Database\Eloquent\Factories\Factory;

class FlightFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = Flight::class;
}
```

Factory States

State manipulation methods allow you to define discrete modifications that can be applied to your model factories in any combination. For example, your `Database\Factories\UserFactory` factory might contain a `suspended` state method that modifies one of its default attribute values.

State transformation methods typically call the `state` method provided by Laravel's base factory class. The `state` method accepts a closure which will receive the array of raw attributes defined for the factory and should return an array of attributes to modify:

```
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * Indicate that the user is suspended.
 */
public function suspended(): Factory
{
    return $this->state(function (array $attributes) {
        return [
            'account_status' => 'suspended',
        ];
    });
}
```

"Trashed" State

If your Eloquent model can be soft deleted, you may invoke the built-in `trashed` state method to indicate that the created model should already be "soft deleted". You do not need to manually define the `trashed` state as it is automatically available to all factories:

```
use App\Models\User;

$user = User::factory()->trashed()->create();
```

Factory Callbacks

Factory callbacks are registered using the `afterMaking` and `afterCreating` methods and allow you to perform additional tasks after making or creating a model. You should register these callbacks by defining a `configure` method on your factory class. This method will be automatically called by Laravel when the factory is instantiated:

```

namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    /**
     * Configure the model factory.
     */
    public function configure(): static
    {
        return $this->afterMaking(function (User $user) {
            // ...
        })->afterCreating(function (User $user) {
            // ...
        });
    }

    // ...
}

```

You may also register factory callbacks within state methods to perform additional tasks that are specific to a given state:

```

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * Indicate that the user is suspended.
 */
public function suspended(): Factory
{
    return $this->state(function (array $attributes) {
        return [
            'account_status' => 'suspended',
        ];
    })->afterMaking(function (User $user) {
        // ...
    })->afterCreating(function (User $user) {
        // ...
    });
}

```

Creating Models Using Factories

Instantiating Models

Once you have defined your factories, you may use the static `factory` method provided to your models by the `Illuminate\Database\Eloquent\Factories\HasFactory` trait in order to instantiate a factory instance for that model. Let's take a look at a few examples of creating models. First, we'll use the `make` method to create models without persisting them to the database:

```
use App\Models\User;

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

You may create a collection of many models using the `count` method:

```
$users = User::factory()->count(3)->make();
```

Applying States

You may also apply any of your `states` to the models. If you would like to apply multiple state transformations to the models, you may simply call the state transformation methods directly:

```
$users = User::factory()->count(5)->suspended()->make();
```

Overriding Attributes

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

```
$user = User::factory()->make([
    'name' => 'Abigail Otwell',
]);
```

Alternatively, the `state` method may be called directly on the factory instance to perform an inline state transformation:

```
$user = User::factory()->state([
    'name' => 'Abigail Otwell',
])->make();
```



Mass assignment protection is automatically disabled when creating models using factories.

Persisting Models

The `create` method instantiates model instances and persists them to the database using Eloquent's `save` method:

```
use App\Models\User;

// Create a single App\Models\User instance...
$user = User::factory()->create();

// Create three App\Models\User instances...
$users = User::factory()->count(3)->create();
```

You may override the factory's default model attributes by passing an array of attributes to the `create` method:

```
$user = User::factory()->create([
    'name' => 'Abigail',
]);
```

Sequences

Sometimes you may wish to alternate the value of a given model attribute for each created model. You may accomplish this by defining a state transformation as a sequence. For example, you may wish to alternate the value of an `admin` column between `Y` and `N` for each created user:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
    ->count(10)
    ->state(new Sequence(
        ['admin' => 'Y'],
        ['admin' => 'N'],
    ))
    ->create();
```

In this example, five users will be created with an `admin` value of `Y` and five users will be created with an `admin` value of `N`.

If necessary, you may include a closure as a sequence value. The closure will be invoked each time the sequence needs a new value:

```
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
    ->count(10)
    ->state(new Sequence(
        fn (Sequence $sequence) => ['role' => UserRoles::all()->random()],
    ))
    ->create();
```

Within a sequence closure, you may access the `$index` or `$count` properties on the sequence instance that is injected into the closure. The `$index` property contains the number of iterations through the sequence that have occurred thus far, while the `$count` property contains the total number of times the sequence will be invoked:

```
$users = User::factory()
    ->count(10)
    ->sequence(fn (Sequence $sequence) => ['name' => "Name ".$sequence->index])
    ->create();
```

For convenience, sequences may also be applied using the `sequence` method, which simply invokes the `state` method internally. The `sequence` method accepts a closure or arrays of sequenced attributes:

```
$users = User::factory()
    ->count(2)
    ->sequence(
        ['name' => 'First User'],
        ['name' => 'Second User'],
    )
    ->create();
```

Factory Relationships

Has Many Relationships

Next, let's explore building Eloquent model relationships using Laravel's fluent factory methods. First, let's assume our application has an `App\Models\User` model and an `App\Models\Post` model. Also, let's assume that the `User` model defines a `hasMany` relationship with `Post`. We can create a user that has three posts using the `has` method provided by the Laravel's factories. The `has` method accepts a factory instance:

```
use App\Models\Post;
use App\Models\User;

$user = User::factory()
    ->has(Post::factory()->count(3))
    ->create();
```

By convention, when passing a `Post` model to the `has` method, Laravel will assume that the `User` model must have a `posts` method that defines the relationship. If necessary, you may explicitly specify the name of the relationship that you would like to manipulate:

```
$user = User::factory()
    ->has(Post::factory()->count(3), 'posts')
    ->create();
```

Of course, you may perform state manipulations on the related models. In addition, you may pass a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
    ->has(
        Post::factory()
            ->count(3)
            ->state(function (array $attributes, User $user) {
                return ['user_type' => $user->type];
            })
    )
    ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to build relationships. For example, the following example will use convention to determine that the related models should be created via a `posts` relationship method on the `User` model:

```
$user = User::factory()
    ->hasPosts(3)
    ->create();
```

When using magic methods to create factory relationships, you may pass an array of attributes to override on the related models:

```
$user = User::factory()
    ->hasPosts(3, [
        'published' => false,
    ])
    ->create();
```

You may provide a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
    ->hasPosts(3, function (array $attributes, User $user) {
        return ['user_type' => $user->type];
    })
    ->create();
```

Belongs To Relationships

Now that we have explored how to build "has many" relationships using factories, let's explore the inverse of the relationship. The `for` method may be used to define the parent model that factory created models belong to. For example, we can create three `App\Models\Post` model instances that belong to a single user:

```
use App\Models\Post;
use App\Models\User;

$post = Post::factory()
    ->count(3)
    ->for(User::factory()->state([
        'name' => 'Jessica Archer',
    )))
    ->create();
```

If you already have a parent model instance that should be associated with the models you are creating, you may pass the model instance to the `for` method:

```
$user = User::factory()->create();

$post = Post::factory()
    ->count(3)
    ->for($user)
    ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define "belongs to" relationships. For example, the following example will use convention to determine that the three posts should belong to the `user` relationship on the `Post` model:

```
$posts = Post::factory()
    ->count(3)
    ->forUser([
        'name' => 'Jessica Archer',
    ])
    ->create();
```

Many to Many Relationships

Like [has many relationships](#), "many to many" relationships may be created using the `has` method:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
    ->has(Role::factory()->count(3))
    ->create();
```

Pivot Table Attributes

If you need to define attributes that should be set on the pivot / intermediate table linking the models, you may use the `hasAttached` method. This method accepts an array of pivot table attribute names and values as its second argument:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
    ->hasAttached(
        Role::factory()->count(3),
        ['active' => true]
    )
    ->create();
```

You may provide a closure based state transformation if your state change requires access to the related model:

```
$user = User::factory()
    ->hasAttached(
        Role::factory()
            ->count(3)
            ->state(function (array $attributes, User $user) {
                return ['name' => $user->name.' Role'];
            }),
        ['active' => true]
    )
    ->create();
```

If you already have model instances that you would like to be attached to the models you are creating, you may pass the model instances to the `hasAttached` method. In this example, the same three roles will be attached to all three users:

```
$roles = Role::factory()->count(3)->create();

$user = User::factory()
    ->count(3)
    ->hasAttached($roles, ['active' => true])
    ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define many to many relationships. For example, the following example will use convention to determine that the related models should be created via a `roles` relationship method on the `User` model:

```
$user = User::factory()
    ->hasRoles(1, [
        'name' => 'Editor'
    ])
    ->create();
```

Polymorphic Relationships

[Polymorphic relationships](#) may also be created using factories. Polymorphic "morph many" relationships are created in the same way as typical "has many" relationships. For example, if an `App\Models\Post` model has a `morphMany` relationship with an `App\Models\Comment` model:

```
use App\Models\Post;

$post = Post::factory()->hasComments(3)->create();
```

Morph To Relationships

Magic methods may not be used to create `morphTo` relationships. Instead, the `for` method must be used directly and the name of the relationship must be explicitly provided. For example, imagine that the `Comment` model has a `commentable` method that defines a `morphTo` relationship. In this situation, we may create three comments that belong to a single post by using the `for` method directly:

```
$comments = Comment::factory()->count(3)->for(
    Post::factory(), 'commentable'
)->create();
```

Polymorphic Many to Many Relationships

Polymorphic "many to many" (`morphToMany` / `morphedByMany`) relationships may be created just like non-polymorphic "many to many" relationships:

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
    ->hasAttached(
        Tag::factory()->count(3),
        ['public' => true]
    )
->create();
```

Of course, the magic `has` method may also be used to create polymorphic "many to many" relationships:

```
$videos = Video::factory()
    ->hasTags(3, ['public' => true])
->create();
```

Defining Relationships Within Factories

To define a relationship within your model factory, you will typically assign a new factory instance to the foreign key of the relationship. This is normally done for the "inverse" relationships such as `belongsTo` and `morphTo` relationships. For example, if you would like to create a new user when creating a post, you may do the following:

```
use App\Models\User;

/**
 * Define the model's default state.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
        'title' => fake()->title(),
        'content' => fake()->paragraph(),
    ];
}
```

If the relationship's columns depend on the factory that defines it you may assign a closure to an attribute. The closure will receive the factory's evaluated attribute array:

```
/**
 * Define the model's default state.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
        'user_type' => function (array $attributes) {
            return User::find($attributes['user_id'])->type;
        },
        'title' => fake()->title(),
        'content' => fake()->paragraph(),
    ];
}
```

Recycling an Existing Model for Relationships

If you have models that share a common relationship with another model, you may use the `recycle` method to ensure a single instance of the related model is recycled for all of the relationships created by the factory.

For example, imagine you have `Airline`, `Flight`, and `Ticket` models, where the ticket belongs to an airline and a flight, and the flight also belongs to an airline. When creating tickets, you will probably want the same airline for both the ticket and the flight, so you may pass an airline instance to the `recycle` method:

```
Ticket::factory()
->recycle(Airline::factory()->create())
->create();
```

You may find the `recycle` method particularly useful if you have models belonging to a common user or team.

The `recycle` method also accepts a collection of existing models. When a collection is provided to the `recycle` method, a random model from the collection will be chosen when the factory needs a model of that type:

```
Ticket::factory()
->recycle($airlines)
->create();
```

Chapter 9

Testing

Testing: Getting Started

- [Introduction](#)
- [Environment](#)
- [Creating Tests](#)
- [Running Tests](#)
 - [Running Tests in Parallel](#)
 - [Reporting Test Coverage](#)
 - [Profiling Tests](#)

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a `phpunit.xml` file is already set up for your application. The framework also ships with convenient helper methods that allow you to expressively test your applications.

By default, your application's `tests` directory contains two directories: `Feature` and `Unit`. Unit tests are tests that focus on a very small, isolated portion of your code. In fact, most unit tests probably focus on a single method. Tests within your "Unit" test directory do not boot your Laravel application and therefore are unable to access your application's database or other framework services.

Feature tests may test a larger portion of your code, including how several objects interact with each other or even a full HTTP request to a JSON endpoint. **Generally, most of your tests should be feature tests. These types of tests provide the most confidence that your system as a whole is functioning as intended.**

An `ExampleTest.php` file is provided in both the `Feature` and `Unit` test directories. After installing a new Laravel application, execute the `vendor/bin/phpunit` or `php artisan test` commands to run your tests.

Environment

When running tests, Laravel will automatically set the [configuration environment](#) to `testing` because of the environment variables defined in the `phpunit.xml` file. Laravel also automatically configures the session and cache to the `array` driver so that no session or cache data will be persisted while testing.

You are free to define other testing environment configuration values as necessary. The `testing` environment variables may be configured in your application's `phpunit.xml` file, but make sure to clear your configuration cache using the `config:clear` Artisan command before running your tests!

The `.env.testing` Environment File

In addition, you may create a `.env.testing` file in the root of your project. This file will be used instead of the `.env` file when running PHPUnit tests or executing Artisan commands with the `--env=testing` option.

The `CreatesApplication` Trait

Laravel includes a `CreatesApplication` trait that is applied to your application's base `TestCase` class. This trait contains a `createApplication` method that bootstraps the Laravel application before running your tests. It's important that you leave this trait at its original location as some features, such as Laravel's parallel testing feature, depend on it.

Creating Tests

To create a new test case, use the `make:test` Artisan command. By default, tests will be placed in the `tests/Feature` directory:

```
php artisan make:test UserTest
```

If you would like to create a test within the `tests/Unit` directory, you may use the `--unit` option when executing the `make:test` command:

```
php artisan make:test UserTest --unit
```

If you would like to create a [Pest PHP](#) test, you may provide the `--pest` option to the `make:test` command:

```
php artisan make:test UserTest --pest
php artisan make:test UserTest --unit --pest
```



Test stubs may be customized using [stub publishing](#).

Once the test has been generated, you may define test methods as you normally would using [PHPUnit](#). To run your tests, execute the `vendor/bin/phpunit` or `php artisan test` command from your terminal:

```
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_basic_test(): void
    {
        $this->assertTrue(true);
    }
}
```



If you define your own `setUp` / `tearDown` methods within a test class, be sure to call the respective `parent::setUp()` / `parent::tearDown()` methods on the parent class. Typically, you should invoke `parent::setUp()` at the start of your own `setUp` method, and `parent::tearDown()` at the end of your `tearDown` method.

Running Tests

As mentioned previously, once you've written tests, you may run them using `phpunit`:

```
./vendor/bin/phpunit
```

In addition to the `phpunit` command, you may use the `test` Artisan command to run your tests. The Artisan test runner provides verbose test reports in order to ease development and debugging:

```
php artisan test
```

Any arguments that can be passed to the `phpunit` command may also be passed to the Artisan `test` command:

```
php artisan test --testsuite=Feature --stop-on-failure
```

Running Tests in Parallel

By default, Laravel and PHPUnit execute your tests sequentially within a single process. However, you may greatly reduce the amount of time it takes to run your tests by running tests simultaneously across multiple processes. To get started, you should install the `brianium/paratest` Composer package as a "dev" dependency. Then, include the `--parallel` option when executing the `test` Artisan command:

```
composer require brianium/paratest --dev
php artisan test --parallel
```

By default, Laravel will create as many processes as there are available CPU cores on your machine. However, you may adjust the number of processes using the `--processes` option:

```
php artisan test --parallel --processes=4
```



When running tests in parallel, some PHPUnit options (such as `--do-not-cache-result`) may not be available.

Parallel Testing and Databases

As long as you have configured a primary database connection, Laravel automatically handles creating and migrating a test database for each parallel process that is running your tests. The test databases will be suffixed with a process token which is unique per process. For example, if you have two parallel test processes, Laravel will create and use `your_db_test_1` and `your_db_test_2` test databases.

By default, test databases persist between calls to the `test` Artisan command so that they can be used again by subsequent `test` invocations. However, you may re-create them using the `--recreate-databases` option:

```
php artisan test --parallel --recreate-databases
```

Parallel Testing Hooks

Occasionally, you may need to prepare certain resources used by your application's tests so they may be safely used by multiple test processes.

Using the `ParallelTesting` facade, you may specify code to be executed on the `setUp` and `tearDown` of a process or test case. The given closures receive the `$token` and `$testCase` variables that contain the process token and the current test case, respectively:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\ParallelTesting;
use Illuminate\Support\ServiceProvider;
use PHPUnit\Framework\TestCase;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        ParallelTesting::setUpProcess(function (int $token) {
            // ...
        });

        ParallelTesting::setUpTestCase(function (int $token, TestCase $testCase) {
            // ...
        });

        // Executed when a test database is created...
        ParallelTesting::setUpTestDatabase(function (string $database, int $token) {
            Artisan::call('db:seed');
        });

        ParallelTesting::tearDownTestCase(function (int $token, TestCase $testCase) {
            // ...
        });

        ParallelTesting::tearDownProcess(function (int $token) {
            // ...
        });
    }
}
```

Accessing the Parallel Testing Token

If you would like to access the current parallel process "token" from any other location in your application's test code, you may use the `token` method. This token is a unique, string identifier for an individual test process and may be used to segment resources across parallel test processes. For example, Laravel automatically appends this token to the end of the test databases created by each parallel testing process:

```
$token = ParallelTesting::token();
```

Reporting Test Coverage



This feature requires [Xdebug](#) or [PCOV](#).

When running your application tests, you may want to determine whether your test cases are actually covering the application code and how much application code is used when running your tests. To accomplish this, you may provide the `--coverage` option when invoking the `test` command:

```
php artisan test --coverage
```

Enforcing a Minimum Coverage Threshold

You may use the `--min` option to define a minimum test coverage threshold for your application. The test suite will fail if this threshold is not met:

```
php artisan test --coverage --min=80.3
```

Profiling Tests

The Artisan test runner also includes a convenient mechanism for listing your application's slowest tests. Invoke the `test` command with the `--profile` option to be presented with a list of your ten slowest tests, allowing you to easily investigate which tests can be improved to speed up your test suite:

```
php artisan test --profile
```

HTTP Tests

- [Introduction](#)
- [Making Requests](#)
 - [Customizing Request Headers](#)
 - [Cookies](#)
 - [Session / Authentication](#)
 - [Debugging Responses](#)
 - [Exception Handling](#)
- [Testing JSON APIs](#)
 - [Fluent JSON Testing](#)
- [Testing File Uploads](#)
- [Testing Views](#)
 - [Rendering Blade and Components](#)
- [Available Assertions](#)
 - [Response Assertions](#)
 - [Authentication Assertions](#)
 - [Validation Assertions](#)

Introduction

Laravel provides a very fluent API for making HTTP requests to your application and examining the responses. For example, take a look at the feature test defined below:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_the_application_returns_a_successful_response(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

The `get` method makes a `GET` request into the application, while the `assertStatus` method asserts that the returned response should have the given HTTP status code. In addition to this simple assertion, Laravel also contains a variety of assertions for inspecting the response headers, content, JSON structure, and more.

Making Requests

To make a request to your application, you may invoke the `get`, `post`, `put`, `patch`, or `delete` methods within your test. These methods do not actually issue a "real" HTTP request to your application. Instead, the entire network request is simulated internally.

Instead of returning an `Illuminate\Http\Response` instance, test request methods return an instance of `Illuminate\Testing\TestResponse`, which provides a [variety of helpful assertions](#) that allow you to inspect your application's responses:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_a_basic_request(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

In general, each of your tests should only make one request to your application. Unexpected behavior may occur if multiple requests are executed within a single test method.



For convenience, the CSRF middleware is automatically disabled when running tests.

Customizing Request Headers

You may use the `withHeaders` method to customize the request's headers before it is sent to the application. This method allows you to add any custom headers you would like to the request:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     */
    public function test_interacting_with_headers(): void
    {
        $response = $this->withHeaders([
            'X-Header' => 'Value',
        ])->post('/user', ['name' => 'Sally']);

        $response->assertStatus(201);
    }
}
```

Cookies

You may use the `withCookie` or `withCookies` methods to set cookie values before making a request. The `withCookie` method accepts a cookie name and value as its two arguments, while the `withCookies` method accepts an array of name / value pairs:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_cookies(): void
    {
        $response = $this->withCookie('color', 'blue')->get('/');

        $response = $this->withCookies([
            'color' => 'blue',
            'name' => 'Taylor',
        ])->get('/');
    }
}
```

Session / Authentication

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

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_the_session(): void
    {
        $response = $this->withSession(['banned' => false])->get('/');
    }
}
```

Laravel's session is typically used to maintain state for the currently authenticated user. Therefore, the `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a [model factory](#) to generate and authenticate a user:

```
<?php

namespace Tests\Feature;

use App\Models\User;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_an_action_that_requires_authentication(): void
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
            ->withSession(['banned' => false])
            ->get('/');
    }
}
```

You may also specify which guard should be used to authenticate the given user by passing the guard name as the second argument to the `actingAs` method. The guard that is provided to the `actingAs` method will also become the default guard for the duration of the test:

```
$this->actingAs($user, 'web')
```

Debugging Responses

After making a test request to your application, the `dump`, `dumpHeaders`, and `dumpSession` methods may be used to examine and debug the response contents:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_basic_test(): void
    {
        $response = $this->get('/');

        $response->dumpHeaders();

        $response->dumpSession();

        $response->dump();
    }
}
```

Alternatively, you may use the `dd`, `ddHeaders`, and `ddSession` methods to dump information about the response and then stop execution:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_basic_test(): void
    {
        $response = $this->get('/');

        $response->ddHeaders();

        $response->ddSession();

        $response->dd();
    }
}
```

Exception Handling

Sometimes you may want to test that your application is throwing a specific exception. To ensure that the exception does not get caught by Laravel's exception handler and returned as an HTTP response, you may invoke the `withoutExceptionHandling` method before making your request:

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

In addition, if you would like to ensure that your application is not utilizing features that have been deprecated by the PHP language or the libraries your application is using, you may invoke the `withoutDeprecationHandling` method before making your request. When deprecation handling is disabled, deprecation warnings will be converted to exceptions, thus causing your test to fail:

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

The `assertThrows` method may be used to assert that code within a given closure throws an exception of the specified type:

```
$this->assertThrows(
    fn () => (new ProcessOrder)->execute(),
    OrderInvalid::class
);
```

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `json`, `getJson`, `postJson`, `putJson`, `patchJson`, `deleteJson`, and `optionsJson` methods may be used to issue JSON requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/api/user` and assert that the expected JSON data was returned:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     */
    public function test_making_an_api_request(): void
    {
        $response = $this->postJson('/api/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

In addition, JSON response data may be accessed as array variables on the response, making it convenient for you to inspect the individual values returned within a JSON response:

```
$this->assertTrue($response['created']);
```



The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

Asserting Exact JSON Matches

As previously mentioned, the `assertJson` method may be used to assert that a fragment of JSON exists within the JSON response. If you would like to verify that a given array **exactly matches** the JSON returned by your application, you should use the `assertExactJson` method:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     */
    public function test_asserting_an_exact_json_match(): void
    {
        $response = $this->postJson('/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

Asserting on JSON Paths

If you would like to verify that the JSON response contains the given data at a specified path, you should use the `assertJsonPath` method:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     */
    public function test_asserting_a_json_paths_value(): void
    {
        $response = $this->postJson('/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertJsonPath('team.owner.name', 'Darian');
    }
}
```

The `assertJsonPath` method also accepts a closure, which may be used to dynamically determine if the assertion should pass:

```
$response->assertJsonPath('team.owner.name', fn (string $name) => strlen($name) >= 3);
```

Fluent JSON Testing

Laravel also offers a beautiful way to fluently test your application's JSON responses. To get started, pass a closure to the `assertJson` method. This closure will be invoked with an instance of `Illuminate\Testing\Fluent\AssertableJson` which can be used to make assertions against the JSON that was returned by your application. The `where` method may be used to make assertions against a particular attribute of the JSON, while the `missing` method may be used to assert that a particular attribute is missing from the JSON:

```
use Illuminate\Testing\Fluent\AssertableJson;

/**
 * A basic functional test example.
 */
public function test_fluent_json(): void
{
    $response = $this->getJson('/users/1');

    $response
        ->assertJson(fn (AssertableJson $json) =>
            $json->where('id', 1)
                ->where('name', 'Victoria Faith')
                ->where('email', fn (string $email) => str($email)->is('victoria@gmail.com'))
                ->whereNot('status', 'pending')
                ->missing('password')
                ->etc()
        );
}
```

Understanding the `etc` Method

In the example above, you may have noticed we invoked the `etc` method at the end of our assertion chain. This method informs Laravel that there may be other attributes present on the JSON object. If the `etc` method is not used, the test will fail if other attributes that you did not make assertions against exist on the JSON object.

The intention behind this behavior is to protect you from unintentionally exposing sensitive information in your JSON responses by forcing you to either explicitly make an assertion against the attribute or explicitly allow additional attributes via the `etc` method.

However, you should be aware that not including the `etc` method in your assertion chain does not ensure that additional attributes are not being added to arrays that are nested within your JSON object. The `etc` method only ensures that no additional attributes exist at the nesting level in which the `etc` method is invoked.

Asserting Attribute Presence / Absence

To assert that an attribute is present or absent, you may use the `has` and `missing` methods:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('data')
        ->missing('message')
);
```

In addition, the `hasAll` and `missingAll` methods allow asserting the presence or absence of multiple attributes simultaneously:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->hasAll(['status', 'data'])
        ->missingAll(['message', 'code'])
);
```

You may use the `hasAny` method to determine if at least one of a given list of attributes is present:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('status')
        ->hasAny('data', 'message', 'code')
);
```

Asserting Against JSON Collections

Often, your route will return a JSON response that contains multiple items, such as multiple users:

```
Route::get('/users', function () {
    return User::all();
});
```

In these situations, we may use the fluent JSON object's `has` method to make assertions against the users included in the response. For example, let's assert that the JSON response contains three users. Next, we'll make some assertions about the first user in the collection using the `first` method. The `first` method accepts a closure which receives another assertable JSON string that we can use to make assertions about the first object in the JSON collection:

```
$response
->assertJson(fn (AssertableJson $json) =>
    $json->has(3)
        ->first(fn (AssertableJson $json) =>
            $json->where('id', 1)
                ->where('name', 'Victoria Faith')
                ->where('email', fn (string $email) => str($email)-
                    >is('victoria@gmail.com'))
                    ->missing('password')
                    ->etc()
            )
        );
);
```

Scoping JSON Collection Assertions

Sometimes, your application's routes will return JSON collections that are assigned named keys:

```
Route::get('/users', function () {
    return [
        'meta' => [...],
        'users' => User::all(),
    ];
})
```

When testing these routes, you may use the `has` method to assert against the number of items in the collection. In addition, you may use the `has` method to scope a chain of assertions:

```
$response
->assertJson(fn (AssertableJson $json) =>
    $json->has('meta')
        ->has('users', 3)
        ->has('users.0', fn (AssertableJson $json) =>
            $json->where('id', 1)
                ->where('name', 'Victoria Faith')
                ->where('email', fn (string $email) => str($email)-
                    >is('victoria@gmail.com'))
                    ->missing('password')
                    ->etc()
            )
        );
);
```

However, instead of making two separate calls to the `has` method to assert against the `users` collection, you may make a single call which provides a closure as its third parameter. When doing so, the closure will automatically be invoked and scoped to the first item in the collection:

```
$response
->assertJson(fn (AssertableJson $json) =>
    $json->has('meta')
        ->has('users', 3, fn (AssertableJson $json) =>
            $json->where('id', 1)
                ->where('name', 'Victoria Faith')
                ->where('email', fn (string $email) => str($email)-
                    >is('victoria@gmail.com'))
                    ->missing('password')
                    ->etc()
            )
        );
);
```

Asserting JSON Types

You may only want to assert that the properties in the JSON response are of a certain type. The `Illuminate\Testing\Fluent\AssertableJson` class provides the `whereType` and `whereAllType` methods for doing just that:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->whereType('id', 'integer')
        ->whereAllType([
            'users.0.name' => 'string',
            'meta' => 'array'
        ])
);
```

You may specify multiple types using the `|` character, or passing an array of types as the second parameter to the `whereType` method. The assertion will be successful if the response value is any of the listed types:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->whereType('name', 'string|null')
        ->whereType('id', ['string', 'integer'])
);
```

The `whereType` and `whereAllType` methods recognize the following types: `string`, `integer`, `double`, `boolean`, `array`, and `null`.

Testing File Uploads

The `Illuminate\Http\UploadedFile` class provides a `fake` method which may be used to generate dummy files or images for testing. This, combined with the `Storage` facade's `fake` method, greatly simplifies the testing of file uploads. For example, you may combine these two features to easily test an avatar upload form:

```
<?php

namespace Tests\Feature;

use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_avatars_can_be_uploaded(): void
    {
        Storage::fake('avatars');

        $file = UploadedFile::fake()->image('avatar.jpg');

        $response = $this->post('/avatar', [
            'avatar' => $file,
        ]);

        Storage::disk('avatars')->assertExists($file->hashName());
    }
}
```

If you would like to assert that a given file does not exist, you may use the `assertMissing` method provided by the `Storage` facade:

```
Storage::fake('avatars');

// ...

Storage::disk('avatars')->assertMissing('missing.jpg');
```

Fake File Customization

When creating files using the `fake` method provided by the `UploadedFile` class, you may specify the width, height, and size of the image (in kilobytes) in order to better test your application's validation rules:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

In addition to creating images, you may create files of any other type using the `create` method:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

If needed, you may pass a `$mimeType` argument to the method to explicitly define the MIME type that should be returned by the file:

```
UploadedFile::fake()->create(
    'document.pdf', $sizeInKilobytes, 'application/pdf'
);
```

Testing Views

Laravel also allows you to render a view without making a simulated HTTP request to the application. To accomplish this, you may call the `view` method within your test. The `view` method accepts the view name and an optional array of data. The method returns an instance of `Illuminate\Testing\TestView`, which offers several methods to conveniently make assertions about the view's contents:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_a_welcome_view_can_be_rendered(): void
    {
        $view = $this->view('welcome', ['name' => 'Taylor']);

        $view->assertSee('Taylor');
    }
}
```

The `TestView` class provides the following assertion methods: `assertSee`, `assertSeeInOrder`, `assertSeeText`, `assertSeeTextInOrder`, `assertDontSee`, and `assertDontSeeText`.

If needed, you may get the raw, rendered view contents by casting the `TestView` instance to a string:

```
$contents = (string) $this->view('welcome');
```

Sharing Errors

Some views may depend on errors shared in the [global error bag provided by Laravel](#). To hydrate the error bag with error messages, you may use the `withViewErrors` method:

```
$view = $this->withViewErrors([
    'name' => ['Please provide a valid name.']
])->view('form');

$view->assertSee('Please provide a valid name.');
```

Rendering Blade and Components

If necessary, you may use the `blade` method to evaluate and render a raw [Blade](#) string. Like the `view` method, the `blade` method returns an instance of `Illuminate\Testing\TestView`:

```
$view = $this->blade(
    '<x-component :name="$name" />',
    ['name' => 'Taylor']
);

$view->assertSee('Taylor');
```

You may use the `component` method to evaluate and render a [Blade component](#). The `component` method returns an instance of `Illuminate\Testing\TestComponent`:

```
$view = $this->component(Profile::class, ['name' => 'Taylor']);

$view->assertSee('Taylor');
```

Available Assertions

Response Assertions

Laravel's `Illuminate\Testing\TestResponse` class provides a variety of custom assertion methods that you may utilize when testing your application. These assertions may be accessed on the response that is returned by the `json`, `get`, `post`, `put`, and `delete` test methods:

assertAccepted	assertJsonMissingExact	assertSeeText
assertBadRequest	assertJsonMissingValidationErrors	assertSeeTextInOrder
assertConflict	assertJsonPath	assertServerError
assertCookie	assertJsonMissingPath	assertServiceUnavailable
assertCookieExpired	assertJsonStructure	assertSessionHas
assertCookieNotExpired	assertJsonValidationErrors	assertSessionHasInput
assertCookieMissing	assertJsonValidationErrorFor	assertSessionHasAll
assertCreated	assertLocation	assertSessionHasErrors
assertDontSee	assertMethodNotAllowed	assertSessionHasErrorsIn
assertDontSeeText	assertMovedPermanently	assertSessionHasNoErrors
assertDownload	assertContent	assertSessionDoesntHaveErrors
assertExactJson	assertNoContent	assertSessionMissing
assertForbidden	assertStreamedContent	assertStatus
assertFound	assertNotFound	assertSuccessful
assertGone	assertOk	assertToManyRequests
assertHeader	assertPaymentRequired	assertUnauthorized
assertHeaderMissing	assertPlainCookie	assertUnprocessable
assertInternalServerError	assertRedirect	assertUnsupportedMediaType
assertJson	assertRedirectContains	assertValid
assertJsonCount	assertRedirectToRoute	assertInvalid
assertJsonFragment	assertRedirectToSignedRoute	assertViewHas
assertJsonIsArray	assertRequestTimeout	assertViewHasAll
assertJsonIsObject	assertSee	assertViewIs
assertJsonMissing	assertSeeInOrder	assertViewMissing

assertBadRequest

Assert that the response has a bad request (400) HTTP status code:

```
$response->assertBadRequest();
```

assertAccepted

Assert that the response has an accepted (202) HTTP status code:

```
$response->assertAccepted();
```

assertConflict

Assert that the response has a conflict (409) HTTP status code:

```
$response->assertConflict();
```

assertCookie

Assert that the response contains the given cookie:

```
$response->assertCookie($cookieName, $value = null);
```

assertCookieExpired

Assert that the response contains the given cookie and it is expired:

```
$response->assertCookieExpired($cookieName);
```

assertCookieNotExpired

Assert that the response contains the given cookie and it is not expired:

```
$response->assertCookieNotExpired($cookieName);
```

assertCookieMissing

Assert that the response does not contain the given cookie:

```
$response->assertCookieMissing($cookieName);
```

assertCreated

Assert that the response has a 201 HTTP status code:

```
$response->assertCreated();
```

assertDontSee

Assert that the given string is not contained within the response returned by the application. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertDontSee($value, $escaped = true);
```

assertDontSeeText

Assert that the given string is not contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. This method will pass the response content to the `strip_tags` PHP function before making the assertion:

```
$response->assertDontSeeText($value, $escaped = true);
```

assertDownload

Assert that the response is a "download". Typically, this means the invoked route that returned the response returned a `Response::download` response, `BinaryFileResponse`, or `Storage::download` response:

```
$response->assertDownload();
```

If you wish, you may assert that the downloadable file was assigned a given file name:

```
$response->assertDownload('image.jpg');
```

assertExactJson

Assert that the response contains an exact match of the given JSON data:

```
$response->assertExactJson(array $data);
```

assertForbidden

Assert that the response has a forbidden (403) HTTP status code:

```
$response->assertForbidden();
```

assertFound

Assert that the response has a found (302) HTTP status code:

```
$response->assertFound();
```

assertGone

Assert that the response has a gone (410) HTTP status code:

```
$response->assertGone();
```

assertHeader

Assert that the given header and value is present on the response:

```
$response->assertHeader($headerName, $value = null);
```

assertHeaderMissing

Assert that the given header is not present on the response:

```
$response->assertHeaderMissing($headerName);
```

assertInternalServerError

Assert that the response has an "Internal Server Error" (500) HTTP status code:

```
$response->assertInternalServerError();
```

assertJson

Assert that the response contains the given JSON data:

```
$response->assertJson(array $data, $strict = false);
```

The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

assertJsonCount

Assert that the response JSON has an array with the expected number of items at the given key:

```
$response->assertJsonCount($count, $key = null);
```

assertJsonFragment

Assert that the response contains the given JSON data anywhere in the response:

```
Route::get('/users', function () {
    return [
        'users' => [
            [
                'name' => 'Taylor Otwell',
            ],
        ],
    ];
});

$response->assertJsonFragment(['name' => 'Taylor Otwell']);
```

assertJsonIsArray

Assert that the response JSON is an array:

```
$response->assertJsonIsArray();
```

assertJsonIsObject

Assert that the response JSON is an object:

```
$response->assertJsonIsObject();
```

assertJsonMissing

Assert that the response does not contain the given JSON data:

```
$response->assertJsonMissing(array $data);
```

assertJsonMissingExact

Assert that the response does not contain the exact JSON data:

```
$response->assertJsonMissingExact(array $data);
```

assertJsonMissingValidationErrors

Assert that the response has no JSON validation errors for the given keys:

```
$response->assertJsonMissingValidationErrors($keys);
```



The more generic `assertValid` method may be used to assert that a response does not have validation errors that were returned as JSON **and** that no errors were flashed to session storage.

assertJsonPath

Assert that the response contains the given data at the specified path:

```
$response->assertJsonPath($path, $expectedValue);
```

For example, if the following JSON response is returned by your application:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the `name` property of the `user` object matches a given value like so:

```
$response->assertJsonPath('user.name', 'Steve Schoger');
```

assertJsonMissingPath

Assert that the response does not contain the given path:

```
$response->assertJsonMissingPath($path);
```

For example, if the following JSON response is returned by your application:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that it does not contain the `email` property of the `user` object:

```
$response->assertJsonMissingPath('user.email');
```

assertJsonStructure

Assert that the response has a given JSON structure:

```
$response->assertJsonStructure(array $structure);
```

For example, if the JSON response returned by your application contains the following data:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the JSON structure matches your expectations like so:

```
$response->assertJsonStructure([
    'user' => [
        'name',
    ]
]);
```

Sometimes, JSON responses returned by your application may contain arrays of objects:

```
{
    "user": [
        {
            "name": "Steve Schoger",
            "age": 55,
            "location": "Earth"
        },
        {
            "name": "Mary Schoger",
            "age": 60,
            "location": "Earth"
        }
    ]
}
```

In this situation, you may use the `*` character to assert against the structure of all of the objects in the array:

```
$response->assertJsonStructure([
    'user' => [
        '*' => [
            'name',
            'age',
            'location'
        ]
    ]
]);
```

assertJsonValidationErrors

Assert that the response has the given JSON validation errors for the given keys. This method should be used when asserting against responses where the validation errors are returned as a JSON structure instead of being flashed to the session:

```
$response->assertJsonValidationErrors(array $data, $responseKey = 'errors');
```



The more generic `assertInvalid` method may be used to assert that a response has validation errors returned as JSON **or** that errors were flashed to session storage.

assertJsonValidationErrorsFor

Assert the response has any JSON validation errors for the given key:

```
$response->assertJsonValidationErrorsFor(string $key, $responseKey = 'errors');
```

assertMethodNotAllowed

Assert that the response has a method not allowed (405) HTTP status code:

```
$response->assertMethodNotAllowed();
```

assertMovedPermanently

Assert that the response has a moved permanently (301) HTTP status code:

```
$response->assertMovedPermanently();
```

assertLocation

Assert that the response has the given URI value in the `Location` header:

```
$response->assertLocation($uri);
```

assertContent

Assert that the given string matches the response content:

```
$response->assertContent($value);
```

assertNoContent

Assert that the response has the given HTTP status code and no content:

```
$response->assertNoContent($status = 204);
```

assertStreamedContent

Assert that the given string matches the streamed response content:

```
$response->assertStreamedContent($value);
```

assertNotFound

Assert that the response has a not found (404) HTTP status code:

```
$response->assertNotFound();
```

assertOk

Assert that the response has a 200 HTTP status code:

```
$response->assertOk();
```

assertPaymentRequired

Assert that the response has a payment required (402) HTTP status code:

```
$response->assertPaymentRequired();
```

assertPlainCookie

Assert that the response contains the given unencrypted cookie:

```
$response->assertPlainCookie($cookieName, $value = null);
```

assertRedirect

Assert that the response is a redirect to the given URI:

```
$response->assertRedirect($uri = null);
```

assertRedirectContains

Assert whether the response is redirecting to a URI that contains the given string:

```
$response->assertRedirectContains($string);
```

assertRedirectToRoute

Assert that the response is a redirect to the given named route:

```
$response->assertRedirectToRoute($name, $parameters = []);
```

assertRedirectToSignedRoute

Assert that the response is a redirect to the given signed route:

```
$response->assertRedirectToSignedRoute($name = null, $parameters = []);
```

assertRequestTimeout

Assert that the response has a request timeout (408) HTTP status code:

```
$response->assertRequestTimeout();
```

assertSee

Assert that the given string is contained within the response. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertSee($value, $escaped = true);
```

assertSeeInOrder

Assert that the given strings are contained in order within the response. This assertion will automatically escape the given strings unless you pass a second argument of `false`:

```
$response->assertSeeInOrder(array $values, $escaped = true);
```

assertSeeText

Assert that the given string is contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. The response content will be passed to the `strip_tags` PHP function before the assertion is made:

```
$response->assertSeeText($value, $escaped = true);
```

assertSeeTextInOrder

Assert that the given strings are contained in order within the response text. This assertion will automatically escape the given strings unless you pass a second argument of `false`. The response content will be passed to the `strip_tags` PHP function before the assertion is made:

```
$response->assertSeeTextInOrder(array $values, $escaped = true);
```

assertServerError

Assert that the response has a server error (≥ 500 , < 600) HTTP status code:

```
$response->assertServerError();
```

assertServiceUnavailable

Assert that the response has a "Service Unavailable" (503) HTTP status code:

```
$response->assertServiceUnavailable();
```

assertSessionHas

Assert that the session contains the given piece of data:

```
$response->assertSessionHas($key, $value = null);
```

If needed, a closure can be provided as the second argument to the `assertSessionHas` method. The assertion will pass if the closure returns `true`:

```
$response->assertSessionHas($key, function (User $value) {
    return $value->name === 'Taylor Otwell';
});
```

assertSessionHasInput

Assert that the session has a given value in the flashed input array:

```
$response->assertSessionHasInput($key, $value = null);
```

If needed, a closure can be provided as the second argument to the `assertSessionHasInput` method. The assertion will pass if the closure returns `true`:

```
use Illuminate\Support\Facades\Crypt;

$response->assertSessionHasInput($key, function (string $value) {
    return Crypt::decryptString($value) === 'secret';
});
```

assertSessionHasAll

Assert that the session contains a given array of key / value pairs:

```
$response->assertSessionHasAll(array $data);
```

For example, if your application's session contains `name` and `status` keys, you may assert that both exist and have the specified values like so:

```
$response->assertSessionHasAll([
    'name' => 'Taylor Otwell',
    'status' => 'active',
]);
```

assertSessionHasErrors

Assert that the session contains an error for the given `$keys`. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key). This method should be used when testing routes that flash validation errors to the session instead of returning them as a JSON structure:

```
$response->assertSessionHasErrors(
    array $keys = [], $format = null, $errorBag = 'default'
);
```

For example, to assert that the `name` and `email` fields have validation error messages that were flashed to the session, you may invoke the `assertSessionHasErrors` method like so:

```
$response->assertSessionHasErrors(['name', 'email']);
```

Or, you may assert that a given field has a particular validation error message:

```
$response->assertSessionHasErrors([
    'name' => 'The given name was invalid.'
]);
```



*The more generic `assertInvalid` method may be used to assert that a response has validation errors returned as JSON **or** that errors were flashed to session storage.*

assertSessionHasErrorsIn

Assert that the session contains an error for the given `$keys` within a specific `error bag`. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key), within the error bag:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

assertSessionHasNoErrors

Assert that the session has no validation errors:

```
$response->assertSessionHasNoErrors();
```

assertSessionDoesntHaveErrors

Assert that the session has no validation errors for the given keys:

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag = 'default');
```



The more generic `assertValid` method may be used to assert that a response does not have validation errors that were returned as JSON **and** that no errors were flashed to session storage.

assertSessionMissing

Assert that the session does not contain the given key:

```
$response->assertSessionMissing($key);
```

assertStatus

Assert that the response has a given HTTP status code:

```
$response->assertStatus($code);
```

assertSuccessful

Assert that the response has a successful (>= 200 and < 300) HTTP status code:

```
$response->assertSuccessful();
```

assertTooManyRequests

Assert that the response has a too many requests (429) HTTP status code:

```
$response->assertTooManyRequests();
```

assertUnauthorized

Assert that the response has an unauthorized (401) HTTP status code:

```
$response->assertUnauthorized();
```

assertUnprocessable

Assert that the response has an unprocessable entity (422) HTTP status code:

```
$response->assertUnprocessable();
```

assertUnsupportedMediaType

Assert that the response has an unsupported media type (415) HTTP status code:

```
$response->assertUnsupportedMediaType();
```

assertValid

Assert that the response has no validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
// Assert that no validation errors are present...
$response->assertValid();

// Assert that the given keys do not have validation errors...
$response->assertValid(['name', 'email']);
```

assertInvalid

Assert that the response has validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
$response->assertInvalid(['name', 'email']);
```

You may also assert that a given key has a particular validation error message. When doing so, you may provide the entire message or only a small portion of the message:

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

assertViewHas

Assert that the response view contains a given piece of data:

```
$response->assertViewHas($key, $value = null);
```

Passing a closure as the second argument to the `assertViewHas` method will allow you to inspect and make assertions against a particular piece of view data:

```
$response->assertViewHas('user', function (User $user) {
    return $user->name === 'Taylor';
});
```

In addition, view data may be accessed as array variables on the response, allowing you to conveniently inspect it:

```
$this->assertEquals('Taylor', $response['name']);
```

assertViewHasAll

Assert that the response view has a given list of data:

```
$response->assertViewHasAll(array $data);
```

This method may be used to assert that the view simply contains data matching the given keys:

```
$response->assertViewHasAll([
    'name',
    'email',
]);
```

Or, you may assert that the view data is present and has specific values:

```
$response->assertViewHasAll([
    'name' => 'Taylor Otwell',
    'email' => 'taylor@example.com',
]);
```

assertViewIs

Assert that the given view was returned by the route:

```
$response->assertViewIs($value);
```

assertViewMissing

Assert that the given data key was not made available to the view returned in the application's response:

```
$response->assertViewMissing($key);
```

Authentication Assertions

Laravel also provides a variety of authentication related assertions that you may utilize within your application's feature tests. Note that these methods are invoked on the test class itself and not the `Illuminate\Testing\TestResponse` instance returned by methods such as `get` and `post`.

assertAuthenticated

Assert that a user is authenticated:

```
$this->assertAuthenticated($guard = null);
```

assertGuest

Assert that a user is not authenticated:

```
$this->assertGuest($guard = null);
```

assertAuthenticatedAs

Assert that a specific user is authenticated:

```
$this->assertAuthenticatedAs($user, $guard = null);
```

Validation Assertions

Laravel provides two primary validation related assertions that you may use to ensure the data provided in your request was either valid or invalid.

assertValid

Assert that the response has no validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
// Assert that no validation errors are present...
$response->assertValid();

// Assert that the given keys do not have validation errors...
$response->assertValid(['name', 'email']);
```

assertInvalid

Assert that the response has validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
$response->assertInvalid(['name', 'email']);
```

You may also assert that a given key has a particular validation error message. When doing so, you may provide the entire message or only a small portion of the message:

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

Console Tests

- [Introduction](#)
- [Success / Failure Expectations](#)
- [Input / Output Expectations](#)
- [Console Events](#)

Introduction

In addition to simplifying HTTP testing, Laravel provides a simple API for testing your application's [custom console commands](#).

Success / Failure Expectations

To get started, let's explore how to make assertions regarding an Artisan command's exit code. To accomplish this, we will use the `artisan` method to invoke an Artisan command from our test. Then, we will use the `assertExitCode` method to assert that the command completed with a given exit code:

```
/**  
 * Test a console command.  
 */  
public function test_console_command(): void  
{  
    $this->artisan('inspire')->assertExitCode(0);  
}
```

You may use the `assertNotExitCode` method to assert that the command did not exit with a given exit code:

```
$this->artisan('inspire')->assertNotExitCode(1);
```

Of course, all terminal commands typically exit with a status code of `0` when they are successful and a non-zero exit code when they are not successful. Therefore, for convenience, you may utilize the `assertSuccessful` and `assertFailed` assertions to assert that a given command exited with a successful exit code or not:

```
$this->artisan('inspire')->assertSuccessful();  
  
$this->artisan('inspire')->assertFailed();
```

Input / Output Expectations

Laravel allows you to easily "mock" user input for your console commands using the `expectsQuestion` method. In addition, you may specify the exit code and text that you expect to be output by the console command using the `assertExitCode` and `expectsOutput` methods. For example, consider the following console command:

```
Artisan::command('question', function () {
    $name = $this->ask('What is your name?');

    $language = $this->choice('Which language do you prefer?', [
        'PHP',
        'Ruby',
        'Python',
    ]);

    $this->line("Your name is '$name' and you prefer '$language'.");
});
```

You may test this command with the following test which utilizes the `expectsQuestion`, `expectsOutput`, `doesntExpectOutput`, `expectsOutputToContain`, `doesntExpectOutputToContain`, and `assertExitCode` methods:

```
/** 
 * Test a console command.
 */
public function test_console_command(): void
{
    $this->artisan('question')
        ->expectsQuestion('What is your name?', 'Taylor Otwell')
        ->expectsQuestion('Which language do you prefer?', 'PHP')
        ->expectsOutput('Your name is Taylor Otwell and you prefer PHP.')
        ->doesntExpectOutput('Your name is Taylor Otwell and you prefer Ruby.')
        ->expectsOutputToContain('Taylor Otwell')
        ->doesntExpectOutputToContain('you prefer Ruby')
        ->assertExitCode(0);
}
```

Confirmation Expectations

When writing a command which expects confirmation in the form of a "yes" or "no" answer, you may utilize the `expectsConfirmation` method:

```
$this->artisan('module:import')
    ->expectsConfirmation('Do you really wish to run this command?', 'no')
    ->assertExitCode(1);
```

Table Expectations

If your command displays a table of information using Artisan's `table` method, it can be cumbersome to write output expectations for the entire table. Instead, you may use the `expectsTable` method. This method accepts the table's headers as its first argument and the table's data as its second argument:

```
$this->artisan('users:all')
    ->expectsTable([
        'ID',
        'Email',
    ], [
        [1, 'taylor@example.com'],
        [2, 'abigail@example.com'],
    ]);
```

Console Events

By default, the `Illuminate\Console\Events\CommandStarting` and `Illuminate\Console\Events\CommandFinished` events are not dispatched while running your application's tests. However, you can enable these events for a given test class by adding the `Illuminate\Foundation\Testing\WithConsoleEvents` trait to the class:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\WithConsoleEvents;
use Tests\TestCase;

class ConsoleEventTest extends TestCase
{
    use WithConsoleEvents;

    // ...
}
```

Laravel Dusk

- [Introduction](#)
- [Installation](#)
 - [Managing ChromeDriver Installations](#)
 - [Using Other Browsers](#)
- [Getting Started](#)
 - [Generating Tests](#)
 - [Resetting the Database After Each Test](#)
 - [Running Tests](#)
 - [Environment Handling](#)
- [Browser Basics](#)
 - [Creating Browsers](#)
 - [Navigation](#)
 - [Resizing Browser Windows](#)
 - [Browser Macros](#)
 - [Authentication](#)
 - [Cookies](#)
 - [Executing JavaScript](#)
 - [Taking a Screenshot](#)
 - [Storing Console Output to Disk](#)
 - [Storing Page Source to Disk](#)
- [Interacting With Elements](#)
 - [Dusk Selectors](#)
 - [Text, Values, and Attributes](#)
 - [Interacting With Forms](#)
 - [Attaching Files](#)
 - [Pressing Buttons](#)
 - [Clicking Links](#)
 - [Using the Keyboard](#)
 - [Using the Mouse](#)
 - [JavaScript Dialogs](#)
 - [Interacting With Inline Frames](#)
 - [Scoping Selectors](#)
 - [Waiting for Elements](#)
 - [Scrolling an Element Into View](#)
- [Available Assertions](#)
- [Pages](#)
 - [Generating Pages](#)
 - [Configuring Pages](#)
 - [Navigating to Pages](#)
 - [Shorthand Selectors](#)
 - [Page Methods](#)
- [Components](#)
 - [Generating Components](#)
 - [Using Components](#)
- [Continuous Integration](#)
 - [Heroku CI](#)
 - [Travis CI](#)
 - [GitHub Actions](#)

- o [Chipper CI](#)

Introduction

[Laravel Dusk](#) provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your local computer. Instead, Dusk uses a standalone [ChromeDriver](#) installation. However, you are free to utilize any other Selenium compatible driver you wish.

Installation

To get started, you should install [Google Chrome](#) and add the `laravel/dusk` Composer dependency to your project:

```
composer require laravel/dusk --dev
```



*If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.*

After installing the Dusk package, execute the `dusk:install` Artisan command. The `dusk:install` command will create a `tests/Browser` directory, an example Dusk test, and install the Chrome Driver binary for your operating system:

```
php artisan dusk:install
```

Next, set the `APP_URL` environment variable in your application's `.env` file. This value should match the URL you use to access your application in a browser.



If you are using [Laravel Sail](#) to manage your local development environment, please also consult the [Sail documentation](#) on [configuring and running Dusk tests](#).

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is installed by Laravel Dusk via the `dusk:install` command, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 86

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all

# Install the version of ChromeDriver that matches the detected version of Chrome / Chromium for
# your OS...
php artisan dusk:chrome-driver --detect
```



Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone `ChromeDriver` installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the `ChromeDriver`:

```
/** 
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 */
public static function prepare(): void
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;

/**
 * Create the RemoteWebDriver instance.
 */
protected function driver(): RemoteWebDriver
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Resetting the Database After Each Test

Most of the tests you write will interact with pages that retrieve data from your application's database; however, your Dusk tests should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which

will not be applicable or available across HTTP requests. Instead, you have two options: the `DatabaseMigrations` trait and the `DatabaseTruncation` trait.

Using Database Migrations

The `DatabaseMigrations` trait will run your database migrations before each test. However, dropping and re-creating your database tables for each test is typically slower than truncating the tables:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

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



SQLite in-memory databases may not be used when executing Dusk tests. Since the browser executes within its own process, it will not be able to access the in-memory databases of other processes.

Using Database Truncation

Before using the `DatabaseTruncation` trait, you must install the `doctrine/dbal` package using the Composer package manager:

```
composer require --dev doctrine/dbal
```

The `DatabaseTruncation` trait will migrate your database on the first test in order to ensure your database tables have been properly created. However, on subsequent tests, the database's tables will simply be truncated - providing a speed boost over re-running all of your database migrations:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseTruncation;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseTruncation;
}
```

By default, this trait will truncate all tables except the `migrations` table. If you would like to customize the tables that should be truncated, you may define a `$tablesToTruncate` property on your test class:

```
/**
 * Indicates which tables should be truncated.
 *
 * @var array
 */
protected $tablesToTruncate = ['users'];
```

Alternatively, you may define an `$exceptTables` property on your test class to specify which tables should be excluded from truncation:

```
/**
 * Indicates which tables should be excluded from truncation.
 *
 * @var array
 */
protected $exceptTables = ['users'];
```

To specify the database connections that should have their tables truncated, you may define a `$connectionsToTruncate` property on your test class:

```
/**
 * Indicates which connections should have their tables truncated.
 *
 * @var array
 */
protected $connectionsToTruncate = ['mysql'];
```

If you would like to execute code before or after database truncation is performed, you may define `beforeTruncatingDatabase` or `afterTruncatingDatabase` methods on your test class:

```
/**
 * Perform any work that should take place before the database has started truncating.
 */
protected function beforeTruncatingDatabase(): void
{
    //
}

/**
 * Perform any work that should take place after the database has finished truncating.
 */
protected function afterTruncatingDatabase(): void
{
    //
}
```

Running Tests

To run your browser tests, execute the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, such as allowing you to only run the tests for a given [group](#):

```
php artisan dusk --group=foo
```



If you are using [Laravel Sail](#) to manage your local development environment, please consult the [Sail](#) documentation on [configuring and running Dusk tests](#).

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 */  
public static function prepare(): void  
{  
    // static::startChromeDriver();  
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class to reflect the correct port:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;  
  
/**  
 * Create the RemoteWebDriver instance.  
 */  
protected function driver(): RemoteWebDriver  
{  
    return RemoteWebDriver::create(  
        'http://localhost:9515', DesiredCapabilities::chrome()  
    );  
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a

.env.dusk.local file.

When running tests, Dusk will back-up your .env file and rename your Dusk environment to .env. Once the tests have completed, your .env file will be restored.

Browser Basics

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, you may call the `browse` method from within your Dusk test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     */
    public function test_basic_example(): void
    {
        $user = User::factory()->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function (Browser $browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'password')
                ->press('Login')
                ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a closure. A browser instance will automatically be passed to this closure by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, simply add more browser arguments to the signature of the closure given to the `browse` method:

```
$this->browse(function (Browser $first, Browser $second) {
    $first->loginAs(User::find(1))
        ->visit('/home')
        ->waitForText('Message');

    $second->loginAs(User::find(2))
        ->visit('/home')
        ->waitForText('Message')
        ->type('message', 'Hey Taylor')
        ->press('Send');

    $first->waitForText('Hey Taylor')
        ->assertSee('Jeffrey Way');
});
```

Navigation

The `visit` method may be used to navigate to a given URI within your application:

```
$browser->visit('/login');
```

You may use the `visitRoute` method to navigate to a named route:

```
$browser->visitRoute('login');
```

You may navigate "back" and "forward" using the `back` and `forward` methods:

```
$browser->back();
$browser->forward();
```

You may use the `refresh` method to refresh the page:

```
$browser->refresh();
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

The `fitContent` method will resize the browser window to match the size of its content:

```
$browser->fitContent();
```

When a test fails, Dusk will automatically resize the browser to fit the content prior to taking a screenshot. You may disable this feature by calling the `disableFitOnFailure` method within your test:

```
$browser->disableFitOnFailure();
```

You may use the `move` method to move the browser window to a different position on your screen:

```
$browser->move($x = 100, $y = 100);
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a [service provider's](#) `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     */
    public function boot(): void
    {
        Browser::macro('scrollToElement', function (string $element = null) {
            $this->script("$(`html, body`).animate({ scrollTop: `#${$element}`.offset().top }, 0);");

            return $this;
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Browser` instance:

```
$this->browse(function (Browser $browser) use ($user) {
    $browser->visit('/pay')
        ->scrollToElement('#credit-card-details')
        ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with your application's login screen during every test. The `loginAs` method accepts a primary key associated with your authenticatable model or an authenticatable model instance:

```
use App\Models\User;
use Laravel\Dusk\Browser;

$this->browse(function (Browser $browser) {
    $browser->loginAs(User::find(1))
        ->visit('/home');
});
```



After using the `loginAs` method, the user session will be maintained for all tests within the file.

Cookies

You may use the `cookie` method to get or set an encrypted cookie's value. By default, all of the cookies created by Laravel are encrypted:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

You may use the `plainCookie` method to get or set an unencrypted cookie's value:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

You may use the `deleteCookie` method to delete the given cookie:

```
$browser->deleteCookie('name');
```

Executing JavaScript

You may use the `script` method to execute arbitrary JavaScript statements within the browser:

```
$browser->script('document.documentElement.scrollTop = 0');

$browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);

$output = $browser->script('return window.location.pathname');
```

Taking a Screenshot

You may use the `screenshot` method to take a screenshot and store it with the given filename. All screenshots will be stored within the `tests/Browser/screenshots` directory:

```
$browser->screenshot('filename');
```

The `responsiveScreenshots` method may be used to take a series of screenshots at various breakpoints:

```
$browser->responsiveScreenshots('filename');
```

Storing Console Output to Disk

You may use the `storeConsoleLog` method to write the current browser's console output to disk with the given filename. Console output will be stored within the `tests/Browser/console` directory:

```
$browser->storeConsoleLog('filename');
```

Storing Page Source to Disk

You may use the `storeSource` method to write the current page's source to disk with the given filename. The page source will be stored within the `tests/Browser/source` directory:

```
$browser->storeSource('filename');
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...
<button>Login</button>
// Test...
$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, when interacting with a Dusk browser, prefix the selector with `@` to manipulate the attached element within your test:

```
// HTML...
<button dusk="login-button">Login</button>
// Test...
$browser->click('@login-button');
```

If desired, you may customize the HTML attribute that the Dusk selector utilizes via the `selectorHtmlAttribute` method. Typically, this method should be called from the `boot` method of your application's `AppServiceProvider`:

```
use Laravel\Dusk\Dusk;
Dusk::selectorHtmlAttribute('data-dusk');
```

Text, Values, and Attributes

Retrieving and Setting Values

Dusk provides several methods for interacting with the current value, display text, and attributes of elements on the page. For example, to get the "value" of an element that matches a given CSS or Dusk selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

You may use the `inputValue` method to get the "value" of an input element that has a given field name:

```
$value = $browser->inputValue('field');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve the value of an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Interacting With Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an `input` or `textarea` field with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
    ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

You can instruct Dusk to type slowly using the `typeSlowly` method. By default, Dusk will pause for 100 milliseconds between key presses. To customize the amount of time between key presses, you may pass the appropriate number of

milliseconds as the third argument to the method:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');
$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

You may use the `appendSlowly` method to append text slowly:

```
$browser->type('tags', 'foo')
->appendSlowly('tags', 'bar', 'baz');
```

Dropdowns

To select a value available on a `select` element, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second argument:

```
$browser->select('size');
```

By providing an array as the second argument to the `select` method, you can instruct the method to select multiple options:

```
$browser->select('categories', ['Art', 'Music']);
```

Checkboxes

To "check" a checkbox input, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');
```

The `uncheck` method may be used to "uncheck" a checkbox input:

```
$browser->uncheck('terms');
```

Radio Buttons

To "select" a `radio` input option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `radio` input with matching `name` and `value` attributes:

```
$browser->radio('size', 'large');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `file` input with a matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```



The `attach` function requires the `Zip` PHP extension to be installed and enabled on your server.

Pressing Buttons

The `press` method may be used to click a button element on the page. The argument given to the `press` method may be either the display text of the button or a CSS / Dusk selector:

```
$browser->press('Login');
```

When submitting forms, many applications disable the form's submission button after it is pressed and then re-enable the button when the form submission's HTTP request is complete. To press a button and wait for the button to be re-enabled, you may use the `pressAndwaitFor` method:

```
// Press the button and wait a maximum of 5 seconds for it to be enabled...
$browser->pressAndwaitFor('Save');

// Press the button and wait a maximum of 1 second for it to be enabled...
$browser->pressAndwaitFor('Save', 1);
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

You may use the `seeLink` method to determine if a link with the given display text is visible on the page:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```



These methods interact with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Using the Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may instruct Dusk to hold modifier keys while entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `swift` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Another valuable use case for the `keys` method is sending a "keyboard shortcut" combination to the primary CSS selector for your application:

```
$browser->keys('.app', ['{command}', 'j']);
```



All modifier keys such as `{command}` are wrapped in `{}` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be [found on GitHub](#).

Fluent Keyboard Interactions

Dusk also provides a `withKeyboard` method, allowing you to fluently perform complex keyboard interactions via the `Laravel\Dusk\Keyboard` class. The `Keyboard` class provides `press`, `release`, `type`, and `pause` methods:

```
use Laravel\Dusk\Keyboard;

$browser->withKeyboard(function (Keyboard $keyboard) {
    $keyboard->press('c')
        ->pause(1000)
        ->release('c')
        ->type(['c', 'e', 'o']);
});
```

Keyboard Macros

If you would like to define custom keyboard interactions that you can easily re-use throughout your test suite, you may use the `macro` method provided by the `Keyboard` class. Typically, you should call this method from a [service provider's boot](#) method:

```
<?php

namespace App\Providers;

use Facebook\WebDriver\WebDriverKeys;
use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Keyboard;
use Laravel\Dusk\OperatingSystem;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     */
    public function boot(): void
    {
        Keyboard::macro('copy', function (string $element = null) {
            $this->type([
                OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTROL, 'c',
            ]);
        });

        return $this;
    }

    Keyboard::macro('paste', function (string $element = null) {
        $this->type([
            OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTROL, 'v',
        ]);
    });

    return $this;
}
}
```

The `macro` function accepts a name as its first argument and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Keyboard` instance:

```
$browser->click('@textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->copy())
->click('@another-textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->paste());
```

Using the Mouse

Clicking on Elements

The `click` method may be used to click on an element matching the given CSS or Dusk selector:

```
$browser->click('.selector');
```

The `clickAtXPath` method may be used to click on an element matching the given XPath expression:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

The `clickAtPoint` method may be used to click on the topmost element at a given pair of coordinates relative to the viewable area of the browser:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

The `doubleClick` method may be used to simulate the double click of a mouse:

```
$browser->doubleClick();  
  
$browser->doubleClick('.selector');
```

The `rightClick` method may be used to simulate the right click of a mouse:

```
$browser->rightClick();  
  
$browser->rightClick('.selector');
```

The `clickAndHold` method may be used to simulate a mouse button being clicked and held down. A subsequent call to the `releaseMouse` method will undo this behavior and release the mouse button:

```
$browser->clickAndHold('.selector');  
  
$browser->clickAndHold()  
    ->pause(1000)  
    ->releaseMouse();
```

The `controlClick` method may be used to simulate the `ctrl+click` event within the browser:

```
$browser->controlClick();  
  
$browser->controlClick('.selector');
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given CSS or Dusk selector:

```
$browser->mouseover('.selector');
```

Drag and Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', $pixels = 10);  
$browser->dragRight('.selector', $pixels = 10);  
$browser->dragUp('.selector', $pixels = 10);  
$browser->dragDown('.selector', $pixels = 10);
```

Finally, you may drag an element by a given offset:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs. For example, you may use the `waitForDialog` method to wait for a JavaScript dialog to appear. This method accepts an optional argument indicating how many seconds to wait for the dialog to appear:

```
$browser->waitForDialog($seconds = null);
```

The `assertDialogOpened` method may be used to assert that a dialog has been displayed and contains the given message:

```
$browser->assertDialogOpened('Dialog message');
```

If the JavaScript dialog contains a prompt, you may use the `typeInDialog` method to type a value into the prompt:

```
$browser->typeInDialog('Hello World');
```

To close an open JavaScript dialog by clicking the "OK" button, you may invoke the `acceptDialog` method:

```
$browser->acceptDialog();
```

To close an open JavaScript dialog by clicking the "Cancel" button, you may invoke the `dismissDialog` method:

```
$browser->dismissDialog();
```

Interacting With Inline Frames

If you need to interact with elements within an iframe, you may use the `withinFrame` method. All element interactions that take place within the closure provided to the `withinFrame` method will be scoped to the context of the specified iframe:

```
$browser->withinFrame('#credit-card-details', function ($browser) {
    $browser->type('input[name="cardnumber"]', '4242424242424242')
        ->type('input[name="exp-date"]', '12/24')
        ->type('input[name="cvc"]', '123');
})->press('Pay');
});
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to accomplish this. All operations performed within the closure given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function (Browser $table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

You may occasionally need to execute assertions outside of the current scope. You may use the `elsewhere` and `elsewhereWhenAvailable` methods to accomplish this:

```
$browser->with('.table', function (Browser $table) {
    // Current scope is `body .table`...

    $browser->elsewhere('.page-title', function (Browser $title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });

    $browser->elsewhereWhenAvailable('.page-title', function (Browser $title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });
});
```

Waiting for Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to become visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you just need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

If you need to pause the test only if a given condition is `true`, use the `pauseIf` method:

```
$browser->pauseIf(App::environment('production'), 1000);
```

Likewise, if you need to pause the test unless a given condition is `true`, you may use the `pauseUnless` method:

```
$browser->pauseUnless(App::environment('testing'), 1000);
```

Waiting for Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS or Dusk selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

You may also wait until the element matching the given selector contains the given text:

```
// Wait a maximum of five seconds for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World');

// Wait a maximum of one second for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

You may also wait until the element matching the given selector is missing from the page:

```
// Wait a maximum of five seconds until the selector is missing...
$browser->waitForTextIn('.selector');

// Wait a maximum of one second until the selector is missing...
$browser->waitForTextIn('.selector', 1);
```

Or, you may wait until the element matching the given selector is enabled or disabled:

```
// Wait a maximum of five seconds until the selector is enabled...
$browser->waitForTextIn('.selector');

// Wait a maximum of one second until the selector is enabled...
$browser->waitForTextIn('.selector', 1);

// Wait a maximum of five seconds until the selector is disabled...
$browser->waitForTextIn('.selector');

// Wait a maximum of one second until the selector is disabled...
$browser->waitForTextIn('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for an element to appear that matches a given selector and then interact with the element. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used to accomplish this. All element operations performed within the given closure will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function (Browser $modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

Waiting for Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

You may use the `waitForText` method to wait until the displayed text has been removed from the page:

```
// Wait a maximum of five seconds for the text to be removed...
$browser->waitForMissingText('Hello World');

// Wait a maximum of one second for the text to be removed...
$browser->waitForMissingText('Hello World', 1);
```

Waiting for Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting for Inputs

The `waitForInput` method may be used to wait until the given input field is visible on the page:

```
// Wait a maximum of five seconds for the input...
$browser->waitForInput($field);

// Wait a maximum of one second for the input...
$browser->waitForInput($field, 1);
```

Waiting on the Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

The `waitForLocation` method can also be used to wait for the current window location to be a fully qualified URL:

```
$browser->waitForLocation('https://example.com/path');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to wait for a page to reload after performing an action, use the `waitForReload` method:

```
use Laravel\Dusk\Browser;

$browser->waitForReload(function (Browser $browser) {
    $browser->press('Submit');
})
->assertSee('Success!');
```

Since the need to wait for the page to reload typically occurs after clicking a button, you may use the `clickAndwaitForReload` method for convenience:

```
$browser->clickAndwaitForReload('.selector')
    ->assertSee('something');
```

Waiting on JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitFor` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitFor('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitFor('App.data.servers.length > 0', 1);
```

Waiting on Vue Expressions

The `waitForVue` and `waitForVueIsNot` methods may be used to wait until a Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitForVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitForVueIsNot('user.name', null, '@user');
```

Waiting for JavaScript Events

The `waitForEvent` method can be used to pause the execution of a test until a JavaScript event occurs:

```
$browser->waitForEvent('load');
```

The event listener is attached to the current scope, which is the `body` element by default. When using a scoped selector, the event listener will be attached to the matching element:

```
$browser->with('iframe', function (Browser $iframe) {
    // Wait for the iframe's load event...
    $iframe->waitForEvent('load');
});
```

You may also provide a selector as the second argument to the `waitForEvent` method to attach the event listener to a specific element:

```
$browser->waitForEvent('load', '.selector');
```

You may also wait for events on the `document` and `window` objects:

```
// Wait until the document is scrolled...
$browser->waitForEvent('scroll', 'document');

// Wait a maximum of five seconds until the window is resized...
$browser->waitForEvent('resize', 'window', 5);
```

Waiting With a Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given closure to return `true`. The `waitUsing` method accepts the maximum number of seconds to wait, the interval at which the closure should be evaluated, the closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Scrolling an Element Into View

Sometimes you may not be able to click on an element because it is outside of the viewable area of the browser. The `scrollIntoView` method will scroll the browser window until the element at the given selector is within the view:

```
$browser->scrollIntoView('.selector')
->click('.selector');
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

assertTitle	assertDontSee	assertAttribute
assertTitleContains	assertSeeln	assertAttributeContains
assertUrls	assertDontSeeln	assertAttributeDoesntContain
assertSchemes	assertSeeAnythingIn	assertAriaAttribute
assertSchemesNot	assertSeeNothingIn	assertDataAttribute
assertHostIs	assertScript	assertVisible
assertHostIsNot	assertSourceHas	assertPresent
assertPortIs	assertSourceMissing	assertNotPresent
assertPortIsNot	assertSeeLink	assertMissing
assertPathBeginsWith	assertDontSeeLink	assertInputPresent
assertPathIs	assertInputValue	assertInputMissing
assertPathIsNot	assertInputValuesNot	assertDialogOpened
assertRoutes	assertChecked	assertEnabled
assertQueryStringHas	assertNotChecked	assertDisabled
assertQueryStringMissing	assertIndeterminate	assertButtonEnabled
assertFragments	assertRadioSelected	assertButtonDisabled
assertFragmentBeginsWith	assertRadioNotSelected	assertFocused
assertFragmentIsNot	assertSelected	assertNotFocused
assertHasCookie	assertNotSelected	assertAuthenticated
assertHasPlainCookie	assertSelectHasOptions	assertGuest
assertCookieMissing	assertSelectMissingOptions	assertAuthenticatedAs
assertPlainCookieMissing	assertSelectHasOption	assertVue
assertCookieValue	assertSelectMissingOption	assertVuelsNot
assertPlainCookieValue	assertValue	assertVueContains
assertSee	assertValueIsNot	assertVueDoesntContain

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith('/home');
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the URL's current hash fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the URL's current hash fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the URL's current hash fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given encrypted cookie is present:

```
$browser->assertHasCookie($name);
```

assertHasPlainCookie

Assert that the given unencrypted cookie is present:

```
$browser->assertHasPlainCookie($name);
```

assertCookieMissing

Assert that the given encrypted cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertPlainCookieMissing

Assert that the given unencrypted cookie is not present:

```
$browser->assertPlainCookieMissing($name);
```

assertCookieValue

Assert that an encrypted cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSeeAnythingIn

Assert that any text is present within the selector:

```
$browser->assertSeeAnythingIn($selector);
```

assertSeeNothingIn

Assert that no text is present within the selector:

```
$browser->assertSeeNothingIn($selector);
```

assertScript

Assert that the given JavaScript expression evaluates to the given value:

```
$browser->assertScript('window.isLoaded')
    ->assertScript('document.readyState', 'complete');
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValuesNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertIndeterminate

Assert that the given checkbox is in an indeterminate state:

```
$browser->assertIndeterminate($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertSelectMissingOption

Assert that the given value is not available to be selected:

```
$browser->assertSelectMissingOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertValueIsNot

Assert that the element matching the given selector does not have the given value:

```
$browser->assertValueIsNot($selector, $value);
```

assertAttribute

Assert that the element matching the given selector has the given value in the provided attribute:

```
$browser->assertAttribute($selector, $attribute, $value);
```

assertAttributeContains

Assert that the element matching the given selector contains the given value in the provided attribute:

```
$browser->assertAttributeContains($selector, $attribute, $value);
```

assertAttributeDoesntContain

Assert that the element matching the given selector does not contain the given value in the provided attribute:

```
$browser->assertAttributeDoesntContain($selector, $attribute, $value);
```

assertAriaAttribute

Assert that the element matching the given selector has the given value in the provided aria attribute:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

For example, given the markup `<button aria-label="Add"></button>`, you may assert against the `aria-label` attribute like so:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

assertDataAttribute

Assert that the element matching the given selector has the given value in the provided data attribute:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

For example, given the markup `<tr id="row-1" data-content="attendees"></tr>`, you may assert against the `data-label` attribute like so:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present in the source:

```
$browser->assertPresent($selector);
```

assertNotPresent

Assert that the element matching the given selector is not present in the source:

```
$browser->assertNotPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertInputPresent

Assert that an input with the given name is present:

```
$browser->assertInputPresent($name);
```

assertInputMissing

Assert that an input with the given name is not present in the source:

```
$browser->assertInputMissing($name);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertAuthenticated

Assert that the user is authenticated:

```
$browser->assertAuthenticated();
```

assertGuest

Assert that the user is not authenticated:

```
$browser->assertGuest();
```

assertAuthenticatedAs

Assert that the user is authenticated as the given user:

```
$browser->assertAuthenticatedAs($user);
```

assertVue

Dusk even allows you to make assertions on the state of [Vue component](#) data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/** 
 * A basic Vue test example.
 */
public function test_vue(): void
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
            ->assertVue('user.name', 'Taylor', '@profile-component');
    });
}
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

assertVueDoesntContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesntContain($property, $value, $componentSelector = null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Dusk Pages allow you to define expressive actions that may then be performed on a given page via a single method. Pages also allow you to define short-cuts to common selectors for your application or for a single page.

Generating Pages

To generate a page object, execute the `dusk:page` Artisan command. All page objects will be placed in your application's `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be [discussed in more detail below](#).

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**  
 * Get the URL for the page.  
 */  
public function url(): string  
{  
    return '/login';  
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. It is not actually necessary to place anything within this method; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**  
 * Assert that the browser is on the page.  
 */  
public function assert(Browser $browser): void  
{  
    $browser->assertPathIs($this->url());  
}
```

Navigating to Pages

Once a page has been defined, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

Shorthand Selectors

The `elements` method within page classes allows you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array<string, string>
 */
public function elements(): array
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Once the shortcut has been defined, you may use the shorthand selector anywhere you would typically use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array<string, string>
 */
public static function siteElements(): array
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     */
    public function createPlaylist(Browser $browser, string $name): void
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed as the first argument to custom page methods:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, execute the `dusk:component` Artisan command. New components are placed in the `tests/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```

<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     */
    public function selector(): string
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     */
    public function assert(Browser $browser): void
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array<string, string>
     */
    public function elements(): array
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     */
    public function selectDate(Browser $browser, int $year, int $month, int $day): void
    {
        $browser->click('@date-field')
            ->within('@year-list', function (Browser $browser) use ($year) {
                $browser->click($year);
            })
            ->within('@month-list', function (Browser $browser) use ($month) {
                $browser->click($month);
            })
            ->within('@day-list', function (Browser $browser) use ($day) {
                $browser->click($day);
            });
    }
}

```

```

    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```

<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     */
    public function test_basic_example(): void
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                ->within(new DatePicker, function (Browser $browser) {
                    $browser->selectDate(2019, 1, 30);
                })
                ->assertSee('January');
        });
    }
}
```

Continuous Integration



Most Dusk continuous integration configurations expect your Laravel application to be served using the built-in PHP development server on port 8000. Therefore, before continuing, you should ensure that your continuous integration environment has an `APP_URL` environment variable value of `http://127.0.0.1:8000`.

Heroku CI

To run Dusk tests on [Heroku CI](#), add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &' && nohup bash -c 'php artisan serve --no-reload > /dev/null 2>&1 && php artisan dusk' "
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on [Travis CI](#), use the following `.travis.yml` configuration. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost &
  - php artisan serve --no-reload &

script:
  - php artisan dusk
```

GitHub Actions

If you are using [GitHub Actions](#) to run your Dusk tests, you may use the following configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```

name: CI
on: [push]
jobs:

dusk-php:
  runs-on: ubuntu-latest
  env:
    APP_URL: "http://127.0.0.1:8000"
    DB_USERNAME: root
    DB_PASSWORD: root
    MAIL_MAILER: log
  steps:
    - uses: actions/checkout@v4
    - name: Prepare The Environment
      run: cp .env.example .env
    - name: Create Database
      run: |
        sudo systemctl start mysql
        mysql --user="root" --password="root" -e "CREATE DATABASE `my-database`\ character
set UTF8mb4 collate utf8mb4_bin;"
    - name: Install Composer Dependencies
      run: composer install --no-progress --prefer-dist --optimize-autoloader
    - name: Generate Application Key
      run: php artisan key:generate
    - name: Upgrade Chrome Driver
      run: php artisan dusk:chrome-driver --detect
    - name: Start Chrome Driver
      run: ./vendor/laravel/dusk/bin/chromedriver-linux &
    - name: Run Laravel Server
      run: php artisan serve --no-reload &
    - name: Run Dusk Tests
      run: php artisan dusk
    - name: Upload Screenshots
      if: failure()
      uses: actions/upload-artifact@v2
      with:
        name: screenshots
        path: tests/Browser/screenshots
    - name: Upload Console Logs
      if: failure()
      uses: actions/upload-artifact@v2
      with:
        name: console
        path: tests/Browser/console

```

Chipper CI

If you are using [Chipper CI](#) to run your Dusk tests, you may use the following configuration file as a starting point. We will use PHP's built-in server to run Laravel so we can listen for requests:

```
# file .chipperci.yml
version: 1

environment:
  php: 8.2
  node: 16

# Include Chrome in the build environment
services:
  - dusk

# Build all commits
on:
  push:
    branches: .*

pipeline:
  - name: Setup
    cmd: |
      cp -v .env.example .env
      composer install --no-interaction --prefer-dist --optimize-autoloader
      php artisan key:generate

      # Create a dusk env file, ensuring APP_URL uses BUILD_HOST
      cp -v .env .env.dusk.ci
      sed -i "s@APP_URL=.*@APP_URL=http://$BUILD_HOST:8000@g" .env.dusk.ci

  - name: Compile Assets
    cmd: |
      npm ci --no-audit
      npm run build

  - name: Browser Tests
    cmd: |
      php -S [::0]:8000 -t public 2>server.log &
      sleep 2
      php artisan dusk:chrome-driver $CHROME_DRIVER
      php artisan dusk --env=ci
```

To learn more about running Dusk tests on Chipper CI, including how to use databases, consult the [official Chipper CI documentation](#).

Database Testing

- [Introduction](#)
 - [Resetting the Database After Each Test](#)
- [Model Factories](#)
- [Running Seeders](#)
- [Available Assertions](#)

Introduction

Laravel provides a variety of helpful tools and assertions to make it easier to test your database driven applications. In addition, Laravel model factories and seeders make it painless to create test database records using your application's Eloquent models and relationships. We'll discuss all of these powerful features in the following documentation.

Resetting the Database After Each Test

Before proceeding much further, let's discuss how to reset your database after each of your tests so that data from a previous test does not interfere with subsequent tests. Laravel's included

`Illuminate\Foundation\Testing\RefreshDatabase` trait will take care of this for you. Simply use the trait on your test class:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     */
    public function test_basic_example(): void
    {
        $response = $this->get('/');

        // ...
    }
}
```

The `Illuminate\Foundation\Testing\RefreshDatabase` trait does not migrate your database if your schema is up to date. Instead, it will only execute the test within a database transaction. Therefore, any records added to the database by test cases that do not use this trait may still exist in the database.

If you would like to totally reset the database, you may use the `Illuminate\Foundation\Testing\DatabaseMigrations` or `Illuminate\Foundation\Testing\DatabaseTruncation` traits instead. However, both of these options are significantly slower than the `RefreshDatabase` trait.

Model Factories

When testing, you may need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Laravel allows you to define a set of default attributes for each of your [Eloquent models](#) using [model factories](#).

To learn more about creating and utilizing model factories to create models, please consult the complete [model factory documentation](#). Once you have defined a model factory, you may utilize the factory within your test to create models:

```
use App\Models\User;

public function test_models_can_be_instantiated(): void
{
    $user = User::factory()->create();

    // ...
}
```

Running Seeders

If you would like to use [database seeders](#) to populate your database during a feature test, you may invoke the `seed` method. By default, the `seed` method will execute the `DatabaseSeeder`, which should execute all of your other seeders. Alternatively, you pass a specific seeder class name to the `seed` method:

```
<?php

namespace Tests\Feature;

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Test creating a new order.
     */
    public function test_orders_can_be_created(): void
    {
        // Run the DatabaseSeeder...
        $this->seed();

        // Run a specific seeder...
        $this->seed(OrderStatusSeeder::class);

        // ...

        // Run an array of specific seeders...
        $this->seed([
            OrderStatusSeeder::class,
            TransactionStatusSeeder::class,
            // ...
        ]);
    }
}
```

Alternatively, you may instruct Laravel to automatically seed the database before each test that uses the `RefreshDatabase` trait. You may accomplish this by defining a `$seed` property on your base test class:

```
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    /**
     * Indicates whether the default seeder should run before each test.
     *
     * @var bool
     */
    protected $seed = true;
}
```

When the `$seed` property is `true`, the test will run the `Database\Seeders\DatabaseSeeder` class before each test that uses the `RefreshDatabase` trait. However, you may specify a specific seeder that should be executed by defining a `$seeder` property on your test class:

```
use Database\Seeders\OrderStatusSeeder;

/**
 * Run a specific seeder before each test.
 *
 * @var string
 */
protected $seeder = OrderStatusSeeder::class;
```

Available Assertions

Laravel provides several database assertions for your [PHPUnit](#) feature tests. We'll discuss each of these assertions below.

assertDatabaseCount

Assert that a table in the database contains the given number of records:

```
$this->assertDatabaseCount('users', 5);
```

assertDatabaseHas

Assert that a table in the database contains records matching the given key / value query constraints:

```
$this->assertDatabaseHas('users', [
    'email' => 'sally@example.com',
]);
```

assertDatabaseMissing

Assert that a table in the database does not contain records matching the given key / value query constraints:

```
$this->assertDatabaseMissing('users', [
    'email' => 'sally@example.com',
]);
```

assertSoftDeleted

The `assertSoftDeleted` method may be used to assert a given Eloquent model has been "soft deleted":

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

assertNotSoftDeleted

The `assertNotSoftDeleted` method may be used to assert a given Eloquent model hasn't been "soft deleted":

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

assertModelExists

Assert that a given model exists in the database:

```
use App\Models\User;

$user = User::factory()->create();

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

assertModelMissing

Assert that a given model does not exist in the database:

```
use App\Models\User;

$user = User::factory()->create();

$user->delete();

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

expectsDatabaseQueryCount

The `expectsDatabaseQueryCount` method may be invoked at the beginning of your test to specify the total number of database queries that you expect to be run during the test. If the actual number of executed queries does not exactly match this expectation, the test will fail:

```
$this->expectsDatabaseQueryCount(5);

// Test...
```

Mocking

- [Introduction](#)
- [Mocking Objects](#)
- [Mocking Facades](#)
 - [Facade Spies](#)
- [Interacting With Time](#)

Introduction

When testing Laravel applications, you may wish to "mock" certain aspects of your application so they are not actually executed during a given test. For example, when testing a controller that dispatches an event, you may wish to mock the event listeners so they are not actually executed during the test. This allows you to only test the controller's HTTP response without worrying about the execution of the event listeners since the event listeners can be tested in their own test case.

Laravel provides helpful methods for mocking events, jobs, and other facades out of the box. These helpers primarily provide a convenience layer over Mockery so you do not have to manually make complicated Mockery method calls.

Mocking Objects

When mocking an object that is going to be injected into your application via Laravel's [service container](#), you will need to bind your mocked instance into the container as an `instance` binding. This will instruct the container to use your mocked instance of the object instead of constructing the object itself:

```
use App\Service;
use Mockery;
use Mockery\MockInterface;

public function test_something_can_be_mocked(): void
{
    $this->instance(
        Service::class,
        Mockery::mock(Service::class, function (MockInterface $mock) {
            $mock->shouldReceive('process')->once();
        })
    );
}
```

In order to make this more convenient, you may use the `mock` method that is provided by Laravel's base test case class. For example, the following example is equivalent to the example above:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->mock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

You may use the `partialMock` method when you only need to mock a few methods of an object. The methods that are not mocked will be executed normally when called:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->partialMock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

Similarly, if you want to `spy` on an object, Laravel's base test case class offers a `spy` method as a convenient wrapper around the `Mockery::spy` method. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

```
use App\Service;

$spy = $this->spy(Service::class);

// ...

$spy->shouldHaveReceived('process');
```

Mocking Facades

Unlike traditional static method calls, `facades` (including `real-time facades`) may be mocked. This provides a great advantage over traditional static methods and grants you the same testability that you would have if you were using traditional dependency injection. When testing, you may often want to mock a call to a Laravel facade that occurs in one of your controllers. For example, consider the following controller action:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

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

        return [
            // ...
        ];
    }
}
```

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

```
<?php

namespace Tests\Feature;

use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

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

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

        // ...
    }
}
```



You should not mock the `Request` facade. Instead, pass the input you desire into the [HTTP testing methods](#) such as `get` and `post` when running your test. Likewise, instead of mocking the `Config` facade, call the `Config::set` method in your tests.

Facade Spies

If you would like to `spy` on a facade, you may call the `spy` method on the corresponding facade. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

```
use Illuminate\Support\Facades\Cache;

public function test_values_are_be_stored_in_cache(): void
{
    Cache::spy();

    $response = $this->get('/');
    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->once()->with(['name' => 'Taylor', 'age' => 10]);
}
```

Interacting With Time

When testing, you may occasionally need to modify the time returned by helpers such as `now` or `Illuminate\Support\Carbon::now()`. Thankfully, Laravel's base feature test class includes helpers that allow you to manipulate the current time:

```

use Illuminate\Support\Carbon;

public function test_time_can_be_manipulated(): void
{
    // Travel into the future...
    $this->travel(5)->milliseconds();
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // Travel into the past...
    $this->travel(-5)->hours();

    // Travel to an explicit time...
    $this->travelTo(now()->subHours(6));

    // Return back to the present time...
    $this->travelBack();
}

```

You may also provide a closure to the various time travel methods. The closure will be invoked with time frozen at the specified time. Once the closure has executed, time will resume as normal:

```

$this->travel(5)->days(function () {
    // Test something five days into the future...
});

$this->travelTo(now()->subDays(10), function () {
    // Test something during a given moment...
});

```

The `freezeTime` method may be used to freeze the current time. Similarly, the `freezeSecond` method will freeze the current time but at the start of the current second:

```

use Illuminate\Support\Carbon;

// Freeze time and resume normal time after executing closure...
$this->freezeTime(function (Carbon $time) {
    // ...
});

// Freeze time at the current second and resume normal time after executing closure...
$this->freezeSecond(function (Carbon $time) {
    // ...
})

```

As you would expect, all of the methods discussed above are primarily useful for testing time sensitive application behavior, such as locking inactive posts on a discussion forum:

```
use App\Models\Thread;

public function test_forum_threads_lock_after_one_week_of_inactivity()
{
    $thread = Thread::factory()->create();

    $this->travel(1)->week();

    $this->assertTrue($thread->isLockedByInactivity());
}
```

Chapter 10

Packages

null

Laravel Cashier (Stripe)

- [Introduction](#)
- [Upgrading Cashier](#)
- [Installation](#)
- [Configuration](#)
 - [Billable Model](#)
 - [API Keys](#)
 - [Currency Configuration](#)
 - [Tax Configuration](#)
 - [Logging](#)
 - [Using Custom Models](#)
- [Quickstart](#)
 - [Selling Products](#)
 - [Selling Subscriptions](#)
- [Customers](#)
 - [Retrieving Customers](#)
 - [Creating Customers](#)
 - [Updating Customers](#)
 - [Balances](#)
 - [Tax IDs](#)
 - [Syncing Customer Data With Stripe](#)
 - [Billing Portal](#)
- [Payment Methods](#)
 - [Storing Payment Methods](#)
 - [Retrieving Payment Methods](#)
 - [Payment Method Presence](#)
 - [Updating the Default Payment Method](#)
 - [Adding Payment Methods](#)
 - [Deleting Payment Methods](#)
- [Subscriptions](#)
 - [Creating Subscriptions](#)
 - [Checking Subscription Status](#)
 - [Changing Prices](#)
 - [Subscription Quantity](#)
 - [Subscriptions With Multiple Products](#)
 - [Multiple Subscriptions](#)
 - [Metered Billing](#)
 - [Subscription Taxes](#)
 - [Subscription Anchor Date](#)
 - [Canceling Subscriptions](#)
 - [Resuming Subscriptions](#)
- [Subscription Trials](#)
 - [With Payment Method Up Front](#)
 - [Without Payment Method Up Front](#)
 - [Extending Trials](#)
- [Handling Stripe Webhooks](#)
 - [Defining Webhook Event Handlers](#)
 - [Verifying Webhook Signatures](#)
- [Single Charges](#)
 - [Simple Charge](#)

- [Charge With Invoice](#)
- [Creating Payment Intents](#)
- [Refunding Charges](#)
- [Checkout](#)
 - [Product Checkouts](#)
 - [Single Charge Checkouts](#)
 - [Subscription Checkouts](#)
 - [Collecting Tax IDs](#)
 - [Guest Checkouts](#)
- [Invoices](#)
 - [Retrieving Invoices](#)
 - [Upcoming Invoices](#)
 - [Previewing Subscription Invoices](#)
 - [Generating Invoice PDFs](#)
- [Handling Failed Payments](#)
 - [Confirming Payments](#)
- [Strong Customer Authentication \(SCA\)](#)
 - [Payments Requiring Additional Confirmation](#)
 - [Off-session Payment Notifications](#)
- [Stripe SDK](#)
- [Testing](#)

Introduction

[Laravel Cashier Stripe](#) provides an expressive, fluent interface to [Stripe's](#) subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading writing. In addition to basic subscription management, Cashier can handle coupons, swapping subscription, subscription "quantities", cancellation grace periods, and even generate invoice PDFs.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review [the upgrade guide](#).



To prevent breaking changes, Cashier uses a fixed Stripe API version. Cashier 15 utilizes Stripe API version `2023-10-16`. The Stripe API version will be updated on minor releases in order to make use of new Stripe features and improvements.

Installation

First, install the Cashier package for Stripe using the Composer package manager:

```
composer require laravel/cashier
```

After installing the package, publish Cashier's migrations using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

Then, migrate your database:

```
php artisan migrate
```

Cashier's migrations will add several columns to your `users` table. They will also create a new `subscriptions` table to hold all of your customer's subscriptions and a `subscription_items` table for subscriptions with multiple prices.

If you wish, you can also publish Cashier's configuration file using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-config"
```

Lastly, to ensure Cashier properly handles all Stripe events, remember to [configure Cashier's webhook handling](#).



Stripe recommends that any column used for storing Stripe identifiers should be case-sensitive. Therefore, you should ensure the column collation for the `stripe_id` column is set to `utf8_bin` when using MySQL. More information regarding this can be found in the [Stripe documentation](#).

Configuration

Billable Model

Before using Cashier, add the `Billable` trait to your billable model definition. Typically, this will be the `App\Models\User` model. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons, and updating payment method information:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

Cashier assumes your billable model will be the `App\Models\User` class that ships with Laravel. If you wish to change this you may specify a different model via the `useCustomerModel` method. This method should typically be called in the `boot` method of your `AppServiceProvider` class:

```
use App\Models\Cashier\User;
use Laravel\Cashier\Cashier;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Cashier::useCustomerModel(User::class);
}
```



If you're using a model other than Laravel's supplied `App\Models\User` model, you'll need to publish and alter the [Cashier migrations](#) provided to match your alternative model's table name.

API Keys

Next, you should configure your Stripe API keys in your application's `.env` file. You can retrieve your Stripe API keys from the Stripe control panel:

```
STRIPE_KEY=your-stripe-key
STRIPE_SECRET=your-stripe-secret
STRIPE_WEBHOOK_SECRET=your-stripe-webhook-secret
```



You should ensure that the `STRIPE_WEBHOOK_SECRET` environment variable is defined in your application's `.env` file, as this variable is used to ensure that incoming webhooks are actually from Stripe.

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by setting the `CASHIER_CURRENCY` environment variable within your application's `.env` file:

```
CASHIER_CURRENCY=eur
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's [NumberFormatter](#) class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```



In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Tax Configuration

Thanks to [Stripe Tax](#), it's possible to automatically calculate taxes for all invoices generated by Stripe. You can enable automatic tax calculation by invoking the `calculateTaxes` method in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use Laravel\Cashier\Cashier;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Cashier::calculateTaxes();
}
```

Once tax calculation has been enabled, any new subscriptions and any one-off invoices that are generated will receive automatic tax calculation.

For this feature to work properly, your customer's billing details, such as the customer's name, address, and tax ID, need to be synced to Stripe. You may use the [customer data synchronization](#) and [Tax ID](#) methods offered by Cashier to accomplish this.



No tax is calculated for [single charges](#) or [single charge checkouts](#).

Logging

Cashier allows you to specify the log channel to be used when logging fatal Stripe errors. You may specify the log channel by defining the `CASHIER_LOGGER` environment variable within your application's `.env` file:

```
CASHIER_LOGGER=stack
```

Exceptions that are generated by API calls to Stripe will be logged through your application's default log channel.

Using Custom Models

You are free to extend the models used internally by Cashier by defining your own model and extending the corresponding Cashier model:

```
use Laravel\Cashier\Subscription as CashierSubscription;

class Subscription extends CashierSubscription
{
    // ...
}
```

After defining your model, you may instruct Cashier to use your custom model via the `Laravel\Cashier\Cashier` class. Typically, you should inform Cashier about your custom models in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use App\Models\Cashier\Subscription;
use App\Models\Cashier\SubscriptionItem;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Cashier::useSubscriptionModel(Subscription::class);
    Cashier::useSubscriptionItemModel(SubscriptionItem::class);
}
```

Quickstart

Selling Products



Before utilizing Stripe Checkout, you should define Products with fixed prices in your Stripe dashboard. In addition, you should [configure Cashier's webhook handling](#).

Offering product and subscription billing via your application can be intimidating. However, thanks to Cashier and [Stripe Checkout](#), you can easily build modern, robust payment integrations.

To charge customers for non-recurring, single-charge products, we'll utilize Cashier to direct customers to Stripe Checkout, where they will provide their payment details and confirm their purchase. Once the payment has been made via Checkout, the customer will be redirected to a success URL of your choosing within your application:

```
use Illuminate\Http\Request;

Route::get('/checkout', function (Request $request) {
    $stripePriceId = 'price_deluxe_album';

    $quantity = 1;

    return $request->user()->checkout([$stripePriceId => $quantity], [
        'success_url' => route('checkout-success'),
        'cancel_url' => route('checkout-cancel'),
    ]);
})->name('checkout');

Route::view('checkout.success')->name('checkout-success');
Route::view('checkout.cancel')->name('checkout-cancel');
```

As you can see in the example above, we will utilize Cashier's provided `checkout` method to redirect the customer to Stripe Checkout for a given "price identifier". When using Stripe, "prices" refer to [defined prices for specific products](#).

If necessary, the `checkout` method will automatically create a customer in Stripe and connect that Stripe customer record to the corresponding user in your application's database. After completing the checkout session, the customer will be redirected to a dedicated success or cancellation page where you can display an informational message to the customer.

Providing Meta Data to Stripe Checkout

When selling products, it's common to keep track of completed orders and purchased products via `Cart` and `Order` models defined by your own application. When redirecting customers to Stripe Checkout to complete a purchase, you may need to provide an existing order identifier so that you can associate the completed purchase with the corresponding order when the customer is redirected back to your application.

To accomplish this, you may provide an array of `metadata` to the `checkout` method. Let's imagine that a pending `Order` is created within our application when a user begins the checkout process. Remember, the `Cart` and `Order` models in this example are illustrative and not provided by Cashier. You are free to implement these concepts based on the needs of your own application:

```
use App\Models\Cart;
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/cart/{cart}/checkout', function (Request $request, Cart $cart) {
    $order = Order::create([
        'cart_id' => $cart->id,
        'price_ids' => $cart->price_ids,
        'status' => 'incomplete',
    ]);

    return $request->user()->checkout($order->price_ids, [
        'success_url' => route('checkout-success').'?session_id={CHECKOUT_SESSION_ID}',
        'cancel_url' => route('checkout-cancel'),
        'metadata' => ['order_id' => $order->id],
    ]);
})->name('checkout');
```

As you can see in the example above, when a user begins the checkout process, we will provide all of the cart / order's associated Stripe price identifiers to the `checkout` method. Of course, your application is responsible for associating these items with the "shopping cart" or order as a customer adds them. We also provide the order's ID to the Stripe Checkout session via the `metadata` array. Finally, we have added the `CHECKOUT_SESSION_ID` template variable to the Checkout success route. When Stripe redirects customers back to your application, this template variable will automatically be populated with the Checkout session ID.

Next, let's build the Checkout success route. This is the route that users will be redirected to after their purchase has been completed via Stripe Checkout. Within this route, we can retrieve the Stripe Checkout session ID and the associated Stripe Checkout instance in order to access our provided meta data and update our customer's order accordingly:

```

use App\Models\Order;
use Illuminate\Http\Request;
use Laravel\Cashier\Cashier;

Route::get('/checkout/success', function (Request $request) {
    $sessionId = $request->get('session_id');

    if ($sessionId === null) {
        return;
    }

    $session = Cashier::stripe()->checkout->sessions->retrieve($sessionId);

    if ($session->payment_status !== 'paid') {
        return;
    }

    $orderId = $session['metadata']['order_id'] ?? null;
    $order = Order::findOrFail($orderId);

    $order->update(['status' => 'completed']);

    return view('checkout-success', ['order' => $order]);
})->name('checkout-success');
}

```

Please refer to Stripe's documentation for more information on the [data contained by the Checkout session object](#).

Selling Subscriptions



Before utilizing Stripe Checkout, you should define Products with fixed prices in your Stripe dashboard. In addition, you should [configure Cashier's webhook handling](#).

Offering product and subscription billing via your application can be intimidating. However, thanks to Cashier and [Stripe Checkout](#), you can easily build modern, robust payment integrations.

To learn how to sell subscriptions using Cashier and Stripe Checkout, let's consider the simple scenario of a subscription service with a basic monthly (`price_basic_monthly`) and yearly (`price_basic_yearly`) plan. These two prices could be grouped under a "Basic" product (`pro_basic`) in our Stripe dashboard. In addition, our subscription service might offer an Expert plan as `pro_expert`.

First, let's discover how a customer can subscribe to our services. Of course, you can imagine the customer might click a "subscribe" button for the Basic plan on our application's pricing page. This button or link should direct the user to a Laravel route which creates the Stripe Checkout session for their chosen plan:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_basic_monthly')
        ->trialDays(5)
        ->allowPromotionCodes()
        ->checkout([
            'success_url' => route('your-success-route'),
            'cancel_url' => route('your-cancel-route'),
        ]);
});
```

As you can see in the example above, we will redirect the customer to a Stripe Checkout session which will allow them to subscribe to our Basic plan. After a successful checkout or cancellation, the customer will be redirected back to the URL we provided to the `checkout` method. To know when their subscription has actually started (since some payment methods require a few seconds to process), we'll also need to [configure Cashier's webhook handling](#).

Now that customers can start subscriptions, we need to restrict certain portions of our application so that only subscribed users can access them. Of course, we can always determine a user's current subscription status via the `subscribed` method provided by Cashier's `Billable` trait:

```
@if ($user->subscribed())
    <p>You are subscribed.</p>
@endif
```

We can even easily determine if a user is subscribed to specific product or price:

```
@if ($user->subscribedToProduct('pro_basic'))
    <p>You are subscribed to our Basic product.</p>
@endif

@if ($user->subscribedToPrice('price_basic_monthly'))
    <p>You are subscribed to our monthly Basic plan.</p>
@endif
```

Building a Subscribed Middleware

For convenience, you may wish to create a [middleware](#) which determines if the incoming request is from a subscribed user. Once this middleware has been defined, you may easily assign it to a route to prevent users that are not subscribed from accessing the route:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class Subscribed
{
    /**
     * Handle an incoming request.
     */
    public function handle(Request $request, Closure $next): Response
    {
        if (! $request->user()?>subscribed()) {
            // Redirect user to billing page and ask them to subscribe...
            return redirect('/billing');
        }

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

Once the middleware has been defined, you may assign it to a route:

```
use App\Http\Middleware\Subscribed;

Route::get('/dashboard', function () {
    // ...
})->middleware([Subscribed::class]);
```

Allowing Customers to Manage Their Billing Plan

Of course, customers may want to change their subscription plan to another product or "tier". The easiest way to allow this is by directing customers to Stripe's [Customer Billing Portal](#), which provides a hosted user interface that allows customers to download invoices, update their payment method, and change subscription plans.

First, define a link or button within your application that directs users to a Laravel route which we will utilize to initiate a Billing Portal session:

```
<a href="{{ route('billing') }}>
    Billing
</a>
```

Next, let's define the route that initiates a Stripe Customer Billing Portal session and redirects the user to the Portal. The `redirectToBillingPortal` method accepts the URL that users should be returned to when exiting the Portal:

```
use Illuminate\Http\Request;

Route::get('/billing', function (Request $request) {
    return $request->user()->redirectToBillingPortal(route('dashboard'));
})->middleware(['auth'])->name('billing');
```



As long as you have configured Cashier's webhook handling, Cashier will automatically keep your application's Cashier-related database tables in sync by inspecting the incoming webhooks from Stripe. So, for example, when a user cancels their subscription via Stripe's Customer Billing Portal, Cashier will receive the corresponding webhook and mark the subscription as "cancelled" in your application's database.

Customers

Retrieving Customers

You can retrieve a customer by their Stripe ID using the `Cashier::findBillable` method. This method will return an instance of the billable model:

```
use Laravel\Cashier\Cashier;

$user = Cashier::findBillable($stripeId);
```

Creating Customers

Occasionally, you may wish to create a Stripe customer without beginning a subscription. You may accomplish this using the `createAsStripeCustomer` method:

```
$stripeCustomer = $user->createAsStripeCustomer();
```

Once the customer has been created in Stripe, you may begin a subscription at a later date. You may provide an optional `$options` array to pass in any additional [customer creation parameters that are supported by the Stripe API](#):

```
$stripeCustomer = $user->createAsStripeCustomer($options);
```

You may use the `asStripeCustomer` method if you want to return the Stripe customer object for a billable model:

```
$stripeCustomer = $user->asStripeCustomer();
```

The `createOrGetStripeCustomer` method may be used if you would like to retrieve the Stripe customer object for a given billable model but are not sure whether the billable model is already a customer within Stripe. This method will create a new customer in Stripe if one does not already exist:

```
$stripeCustomer = $user->createOrGetStripeCustomer();
```

Updating Customers

Occasionally, you may wish to update the Stripe customer directly with additional information. You may accomplish this using the `updateStripeCustomer` method. This method accepts an array of [customer update options supported by the Stripe API](#):

```
$stripeCustomer = $user->updateStripeCustomer($options);
```

Balances

Stripe allows you to credit or debit a customer's "balance". Later, this balance will be credited or debited on new invoices. To check the customer's total balance you may use the `balance` method that is available on your billable model. The `balance` method will return a formatted string representation of the balance in the customer's currency:

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

To credit a customer's balance, you may provide a value to the `creditBalance` method. If you wish, you may also provide a description:

```
$user->creditBalance(500, 'Premium customer top-up.');
```

Providing a value to the `debitBalance` method will debit the customer's balance:

```
$user->debitBalance(300, 'Bad usage penalty.');
```

The `applyBalance` method will create new customer balance transactions for the customer. You may retrieve these transaction records using the `balanceTransactions` method, which may be useful in order to provide a log of credits and debits for the customer to review:

```
// Retrieve all transactions...
$transactions = $user->balanceTransactions();

foreach ($transactions as $transaction) {
    // Transaction amount...
    $amount = $transaction->amount(); // $2.31

    // Retrieve the related invoice when available...
    $invoice = $transaction->invoice();
}
```

Tax IDs

Cashier offers an easy way to manage a customer's tax IDs. For example, the `taxIds` method may be used to retrieve all of the [tax IDs](#) that are assigned to a customer as a collection:

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

You can also retrieve a specific tax ID for a customer by its identifier:

```
$taxId = $user->findTaxId('txi_belgium');
```

You may create a new Tax ID by providing a valid `type` and value to the `createTaxId` method:

```
$taxId = $user->createTaxId('eu_vat', 'BE0123456789');
```

The `createTaxId` method will immediately add the VAT ID to the customer's account. [Verification of VAT IDs is also done by Stripe](#); however, this is an asynchronous process. You can be notified of verification updates by subscribing to the `customer.tax_id.updated` webhook event and inspecting the VAT IDs `verification` parameter. For more information on handling webhooks, please consult the [documentation on defining webhook handlers](#).

You may delete a tax ID using the `deleteTaxId` method:

```
$user->deleteTaxId('txi_belgium');
```

Syncing Customer Data With Stripe

Typically, when your application's users update their name, email address, or other information that is also stored by Stripe, you should inform Stripe of the updates. By doing so, Stripe's copy of the information will be in sync with your application's.

To automate this, you may define an event listener on your billable model that reacts to the model's `updated` event. Then, within your event listener, you may invoke the `syncStripeCustomerDetails` method on the model:

```
use App\Models\User;
use function Illuminate\Events\queueable;

/**
 * The "booted" method of the model.
 */
protected static function booted(): void
{
    static::updated(queueable(function (User $customer) {
        if ($customer->hasStripeId()) {
            $customer->syncStripeCustomerDetails();
        }
    }));
}
```

Now, every time your customer model is updated, its information will be synced with Stripe. For convenience, Cashier will automatically sync your customer's information with Stripe on the initial creation of the customer.

You may customize the columns used for syncing customer information to Stripe by overriding a variety of methods provided by Cashier. For example, you may override the `stripeName` method to customize the attribute that should be considered the customer's "name" when Cashier syncs customer information to Stripe:

```
/**
 * Get the customer name that should be synced to Stripe.
 */
public function stripeName(): string|null
{
    return $this->company_name;
}
```

Similarly, you may override the `stripeEmail`, `stripePhone`, `stripeAddress`, and `stripePreferredLocales` methods. These methods will sync information to their corresponding customer parameters when [updating the Stripe customer object](#). If you wish to take total control over the customer information sync process, you may override the `syncStripeCustomerDetails` method.

Billing Portal

Stripe offers [an easy way to set up a billing portal](#) so that your customer can manage their subscription, payment methods, and view their billing history. You can redirect your users to the billing portal by invoking the `redirectToBillingPortal` method on the billable model from a controller or route:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal();
});
```

By default, when the user is finished managing their subscription, they will be able to return to the `home` route of your application via a link within the Stripe billing portal. You may provide a custom URL that the user should return to by passing the URL as an argument to the `redirectToBillingPortal` method:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal(route('billing'));
});
```

If you would like to generate the URL to the billing portal without generating an HTTP redirect response, you may invoke the `billingPortalUrl` method:

```
$url = $request->user()->billingPortalUrl(route('billing'));
```

Payment Methods

Storing Payment Methods

In order to create subscriptions or perform "one-off" charges with Stripe, you will need to store a payment method and retrieve its identifier from Stripe. The approach used to accomplish this differs based on whether you plan to use the payment method for subscriptions or single charges, so we will examine both below.

Payment Methods for Subscriptions

When storing a customer's credit card information for future use by a subscription, the Stripe "Setup Intents" API must be used to securely gather the customer's payment method details. A "Setup Intent" indicates to Stripe the intention to charge a customer's payment method. Cashier's `Billable` trait includes the `createSetupIntent` method to easily create a new Setup Intent. You should invoke this method from the route or controller that will render the form which gathers your customer's payment method details:

```
return view('update-payment-method', [
    'intent' => $user->createSetupIntent()
]);
```

After you have created the Setup Intent and passed it to the view, you should attach its secret to the element that will gather the payment method. For example, consider this "update payment method" form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button" data-secret="{{ $intent->client_secret }}">
    Update Payment Method
</button>
```

Next, the Stripe.js library may be used to attach a [Stripe Element](#) to the form and securely gather the customer's payment details:

```
<script src="https://js.stripe.com/v3/"></script>

<script>
  const stripe = Stripe('stripe-public-key');

  const elements = stripe.elements();
  const cardElement = elements.create('card');

  cardElement.mount('#card-element');
</script>
```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using [Stripe's confirmCardSetup method](#):

```
const cardHolderName = document.getElementById('card-holder-name');
const cardButton = document.getElementById('card-button');
const clientSecret = cardButton.dataset.secret;

cardButton.addEventListener('click', async (e) => {
  const { setupIntent, error } = await stripe.confirmCardSetup(
    clientSecret,
    {
      payment_method: {
        card: cardElement,
        billing_details: { name: cardHolderName.value }
      }
    }
  );

  if (error) {
    // Display "error.message" to the user...
  } else {
    // The card has been verified successfully...
  }
});
```

After the card has been verified by Stripe, you may pass the resulting `setupIntent.payment_method` identifier to your Laravel application, where it can be attached to the customer. The payment method can either be [added as a new payment method](#) or [used to update the default payment method](#). You can also immediately use the payment method identifier to [create a new subscription](#).



If you would like more information about Setup Intents and gathering customer payment details please [review this overview provided by Stripe](#).

Payment Methods for Single Charges

Of course, when making a single charge against a customer's payment method, we will only need to use a payment method identifier once. Due to Stripe limitations, you may not use the stored default payment method of a customer for single charges. You must allow the customer to enter their payment method details using the Stripe.js library. For example, consider the following form:

```

<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button">
    Process Payment
</button>

```

After defining such a form, the Stripe.js library may be used to attach a [Stripe Element](#) to the form and securely gather the customer's payment details:

```

<script src="https://js.stripe.com/v3/"></script>

<script>
    const stripe = Stripe('stripe-public-key');

    const elements = stripe.elements();
    const cardElement = elements.create('card');

    cardElement.mount('#card-element');
</script>

```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using [Stripe's createPaymentMethod](#) method:

```

const cardHolderName = document.getElementById('card-holder-name');
const cardButton = document.getElementById('card-button');

cardButton.addEventListener('click', async (e) => {
    const { paymentMethod, error } = await stripe.createPaymentMethod(
        'card', cardElement, {
            billing_details: { name: cardHolderName.value }
        }
    );

    if (error) {
        // Display "error.message" to the user...
    } else {
        // The card has been verified successfully...
    }
});

```

If the card is verified successfully, you may pass the `paymentMethod.id` to your Laravel application and process a [single charge](#).

Retrieving Payment Methods

The `paymentMethods` method on the billable model instance returns a collection of [Laravel\Cashier\PaymentMethod](#) instances:

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

By default, this method will return payment methods of every type. To retrieve payment methods of a specific type, you may pass the `type` as an argument to the method:

```
$paymentMethods = $user->paymentMethods('sepa_debit');
```

To retrieve the customer's default payment method, the `defaultPaymentMethod` method may be used:

```
$paymentMethod = $user->defaultPaymentMethod();
```

You can retrieve a specific payment method that is attached to the billable model using the `findPaymentMethod` method:

```
$paymentMethod = $user->findPaymentMethod($paymentMethodId);
```

Payment Method Presence

To determine if a billable model has a default payment method attached to their account, invoke the `hasDefaultPaymentMethod` method:

```
if ($user->hasDefaultPaymentMethod()) {
    // ...
}
```

You may use the `hasPaymentMethod` method to determine if a billable model has at least one payment method attached to their account:

```
if ($user->hasPaymentMethod()) {
    // ...
}
```

This method will determine if the billable model has any payment method at all. To determine if a payment method of a specific type exists for the model, you may pass the `type` as an argument to the method:

```
if ($user->hasPaymentMethod('sepa_debit')) {
    // ...
}
```

Updating the Default Payment Method

The `updateDefaultPaymentMethod` method may be used to update a customer's default payment method information. This method accepts a Stripe payment method identifier and will assign the new payment method as the default billing payment method:

```
$user->updateDefaultPaymentMethod($paymentMethod);
```

To sync your default payment method information with the customer's default payment method information in Stripe, you may use the `updateDefaultPaymentMethodFromStripe` method:

```
$user->updateDefaultPaymentMethodFromStripe();
```



The default payment method on a customer can only be used for invoicing and creating new subscriptions. Due to limitations imposed by Stripe, it may not be used for single charges.

Adding Payment Methods

To add a new payment method, you may call the `addPaymentMethod` method on the billable model, passing the payment method identifier:

```
$user->addPaymentMethod($paymentMethod);
```



To learn how to retrieve payment method identifiers please review the [payment method storage documentation](#).

Deleting Payment Methods

To delete a payment method, you may call the `delete` method on the `Laravel\Cashier\PaymentMethod` instance you wish to delete:

```
$paymentMethod->delete();
```

The `deletePaymentMethod` method will delete a specific payment method from the billable model:

```
$user->deletePaymentMethod('pm_visa');
```

The `deletePaymentMethods` method will delete all of the payment method information for the billable model:

```
$user->deletePaymentMethods();
```

By default, this method will delete payment methods of every type. To delete payment methods of a specific type you can pass the `type` as an argument to the method:

```
$user->deletePaymentMethods('sepa_debit');
```



If a user has an active subscription, your application should not allow them to delete their default payment method.

Subscriptions

Subscriptions provide a way to set up recurring payments for your customers. Stripe subscriptions managed by Cashier provide support for multiple subscription prices, subscription quantities, trials, and more.

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\Models\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription(
        'default', 'price_monthly'
    )->create($request->paymentMethodId);

    // ...
});
```

The first argument passed to the `newSubscription` method should be the internal type of the subscription. If your application only offers a single subscription, you might call this `default` or `primary`. This subscription type is only for internal application usage and is not meant to be shown to users. In addition, it should not contain spaces and it should never be changed after creating the subscription. The second argument is the specific price the user is subscribing to. This value should correspond to the price's identifier in Stripe.

The `create` method, which accepts [a Stripe payment method identifier](#) or Stripe `PaymentMethod` object, will begin the subscription as well as update your database with the billable model's Stripe customer ID and other relevant billing information.



Passing a payment method identifier directly to the `create` subscription method will also automatically add it to the user's stored payment methods.

Collecting Recurring Payments via Invoice Emails

Instead of collecting a customer's recurring payments automatically, you may instruct Stripe to email an invoice to the customer each time their recurring payment is due. Then, the customer may manually pay the invoice once they receive it. The customer does not need to provide a payment method up front when collecting recurring payments via invoices:

```
$user->newSubscription('default', 'price_monthly')->createAndSendInvoice();
```

The amount of time a customer has to pay their invoice before their subscription is cancelled is determined by the `days_until_due` option. By default, this is 30 days; however, you may provide a specific value for this option if you wish:

```
$user->newSubscription('default', 'price_monthly')->createAndSendInvoice([], [
    'days_until_due' => 30
]);
```

Quantities

If you would like to set a specific `quantity` for the price when creating the subscription, you should invoke the `quantity` method on the subscription builder before creating the subscription:

```
$user->newSubscription('default', 'price_monthly')
    ->quantity(5)
    ->create($paymentMethod);
```

Additional Details

If you would like to specify additional [customer](#) or [subscription](#) options supported by Stripe, you may do so by passing them as the second and third arguments to the `create` method:

```
$user->newSubscription('default', 'price_monthly')->create($paymentMethod, [
    'email' => $email,
], [
    'metadata' => ['note' => 'Some extra information.'],
]);

```

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$user->newSubscription('default', 'price_monthly')
    ->withCoupon('code')
    ->create($paymentMethod);
```

Or, if you would like to apply a [Stripe promotion code](#), you may use the `withPromotionCode` method:

```
$user->newSubscription('default', 'price_monthly')
    ->withPromotionCode('promo_code_id')
    ->create($paymentMethod);
```

The given promotion code ID should be the Stripe API ID assigned to the promotion code and not the customer facing promotion code. If you need to find a promotion code ID based on a given customer facing promotion code, you may use the `findPromotionCode` method:

```
// Find a promotion code ID by its customer facing code...
$promotionCode = $user->findPromotionCode('SUMMERSALE');

// Find an active promotion code ID by its customer facing code...
$promotionCode = $user->findActivePromotionCode('SUMMERSALE');
```

In the example above, the returned `$promotionCode` object is an instance of [Laravel\Cashier\PromotionCode](#). This class decorates an underlying [Stripe\PromotionCode](#) object. You can retrieve the coupon related to the promotion code by invoking the `coupon` method:

```
$coupon = $user->findPromotionCode('SUMMERSALE')->coupon();
```

The coupon instance allows you to determine the discount amount and whether the coupon represents a fixed discount or percentage based discount:

```
if ($coupon->isPercentage()) {
    return $coupon->percentOff().'%'; // 21.5%
} else {
    return $coupon->amountOff(); // $5.99
}
```

You can also retrieve the discounts that are currently applied to a customer or subscription:

```
$discount = $billable->discount();

$discount = $subscription->discount();
```

The returned `Laravel\Cashier\Discount` instances decorate an underlying `Stripe\Discount` object instance. You may retrieve the coupon related to this discount by invoking the `coupon` method:

```
$coupon = $subscription->discount()->coupon();
```

If you would like to apply a new coupon or promotion code to a customer or subscription, you may do so via the `applyCoupon` or `applyPromotionCode` methods:

```
$billable->applyCoupon('coupon_id');
$billable->applyPromotionCode('promotion_code_id');

$subscription->applyCoupon('coupon_id');
$subscription->applyPromotionCode('promotion_code_id');
```

Remember, you should use the Stripe API ID assigned to the promotion code and not the customer facing promotion code. Only one coupon or promotion code can be applied to a customer or subscription at a given time.

For more info on this subject, please consult the Stripe documentation regarding [coupons](#) and [promotion codes](#).

Adding Subscriptions

If you would like to add a subscription to a customer who already has a default payment method you may invoke the `add` method on the subscription builder:

```
use App\Models\User;

$user = User::find(1);

$user->newSubscription('default', 'price_monthly')->add();
```

Creating Subscriptions From the Stripe Dashboard

You may also create subscriptions from the Stripe dashboard itself. When doing so, Cashier will sync newly added subscriptions and assign them a type of `default`. To customize the subscription type that is assigned to dashboard created subscriptions, [define webhook event handlers](#).

In addition, you may only create one type of subscription via the Stripe dashboard. If your application offers multiple subscriptions that use different types, only one type of subscription may be added through the Stripe dashboard.

Finally, you should always make sure to only add one active subscription per type of subscription offered by your application. If a customer has two `default` subscriptions, only the most recently added subscription will be used by Cashier even though both would be synced with your application's database.

Checking Subscription Status

Once a customer is subscribed to your application, you may easily check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the customer has an active subscription, even if the subscription is currently within its trial period. The `subscribed` method accepts the type of the subscription as its first argument:

```
if ($user->subscribed('default')) {
    // ...
}
```

The `subscribed` method also makes a great candidate for a [route middleware](#), allowing you to filter access to routes and controllers based on the user's subscription status:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureUserIsSubscribed
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->user() && ! $request->user()->subscribed('default')) {
            // This user is not a paying customer...
            return redirect('billing');
        }

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

If you would like to determine if a user is still within their trial period, you may use the `onTrial` method. This method can be useful for determining if you should display a warning to the user that they are still on their trial period:

```
if ($user->subscription('default')->onTrial()) {
    // ...
}
```

The `subscribedToProduct` method may be used to determine if the user is subscribed to a given product based on a given Stripe product's identifier. In Stripe, products are collections of prices. In this example, we will determine if the user's `default` subscription is actively subscribed to the application's "premium" product. The given Stripe product identifier should correspond to one of your product's identifiers in the Stripe dashboard:

```
if ($user->subscribedToProduct('prod_premium', 'default')) {
    // ...
}
```

By passing an array to the `subscribedToProduct` method, you may determine if the user's `default` subscription is actively subscribed to the application's "basic" or "premium" product:

```
if ($user->subscribedToProduct(['prod_basic', 'prod_premium'], 'default')) {
    // ...
}
```

The `subscribedToPrice` method may be used to determine if a customer's subscription corresponds to a given price ID:

```
if ($user->subscribedToPrice('price_basic_monthly', 'default')) {
    // ...
}
```

The `recurring` method may be used to determine if the user is currently subscribed and is no longer within their trial period:

```
if ($user->subscription('default')->recurring()) {
    // ...
}
```



If a user has two subscriptions with the same type, the most recent subscription will always be returned by the `subscription` method. For example, a user might have two subscription records with the type of `default`; however, one of the subscriptions may be an old, expired subscription, while the other is the current, active subscription. The most recent subscription will always be returned while older subscriptions are kept in the database for historical review.

Canceled Subscription Status

To determine if the user was once an active subscriber but has canceled their subscription, you may use the `canceled` method:

```
if ($user->subscription('default')->canceled()) {
    // ...
}
```

You may also determine if a user has canceled their subscription but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. Note that the `subscribed` method still returns `true` during this time:

```
if ($user->subscription('default')->onGracePeriod()) {
    // ...
}
```

To determine if the user has canceled their subscription and is no longer within their "grace period", you may use the `ended` method:

```
if ($user->subscription('default')->ended()) {
    // ...
}
```

Incomplete and Past Due Status

If a subscription requires a secondary payment action after creation the subscription will be marked as `incomplete`. Subscription statuses are stored in the `stripe_status` column of Cashier's `subscriptions` database table.

Similarly, if a secondary payment action is required when swapping prices the subscription will be marked as `past_due`. When your subscription is in either of these states it will not be active until the customer has confirmed their payment. Determining if a subscription has an incomplete payment may be accomplished using the `hasIncompletePayment` method on the billable model or a subscription instance:

```
if ($user->hasIncompletePayment('default')) {
    // ...
}

if ($user->subscription('default')->hasIncompletePayment()) {
    // ...
}
```

When a subscription has an incomplete payment, you should direct the user to Cashier's payment confirmation page, passing the `latestPayment` identifier. You may use the `latestPayment` method available on subscription instance to retrieve this identifier:

```
<a href="{{ route('cashier.payment', $subscription->latestPayment()->id) }}>
    Please confirm your payment.
</a>
```

If you would like the subscription to still be considered active when it's in a `past_due` or `incomplete` state, you may use the `keepPastDueSubscriptionsActive` and `keepIncompleteSubscriptionsActive` methods provided by Cashier. Typically, these methods should be called in the `register` method of your `App\Providers\AppServiceProvider`:

```
use Laravel\Cashier\Cashier;

/**
 * Register any application services.
 */
public function register(): void
{
    Cashier::keepPastDueSubscriptionsActive();
    Cashier::keepIncompleteSubscriptionsActive();
}
```



When a subscription is in an `incomplete` state it cannot be changed until the payment is confirmed. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in an `incomplete` state.

Subscription Scopes

Most subscription states are also available as query scopes so that you may easily query your database for subscriptions that are in a given state:

```
// Get all active subscriptions...
$subscriptions = Subscription::query()->active()->get();

// Get all of the canceled subscriptions for a user...
$subscriptions = $user->subscriptions()->canceled()->get();
```

A complete list of available scopes is available below:

```
Subscription::query()->active();
Subscription::query()->canceled();
Subscription::query()->ended();
Subscription::query()->incomplete();
Subscription::query()->notCanceled();
Subscription::query()->notOnGracePeriod();
Subscription::query()->notOnTrial();
Subscription::query()->onGracePeriod();
Subscription::query()->onTrial();
Subscription::query()->pastDue();
Subscription::query()->recurring();
```

Changing Prices

After a customer is subscribed to your application, they may occasionally want to change to a new subscription price. To swap a customer to a new price, pass the Stripe price's identifier to the `swap` method. When swapping prices, it is assumed that the user would like to re-activate their subscription if it was previously canceled. The given price identifier should correspond to a Stripe price identifier available in the Stripe dashboard:

```
use App\Models\User;

$user = App\Models\User::find(1);

$user->subscription('default')->swap('price_yearly');
```

If the customer is on trial, the trial period will be maintained. Additionally, if a "quantity" exists for the subscription, that quantity will also be maintained.

If you would like to swap prices and cancel any trial period the customer is currently on, you may invoke the `skipTrial` method:

```
$user->subscription('default')
    ->skipTrial()
    ->swap('price_yearly');
```

If you would like to swap prices and immediately invoice the customer instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = User::find(1);

$user->subscription('default')->swapAndInvoice('price_yearly');
```

Prorations

By default, Stripe prorates charges when swapping between prices. The `noProrate` method may be used to update the subscription's price without prorating the charges:

```
$user->subscription('default')->noProrate()->swap('price_yearly');
```

For more information on subscription proration, consult the [Stripe documentation](#).



Executing the `noProrate` method before the `swapAndInvoice` method will have no effect on proration. An invoice will always be issued.

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, a project management application might charge \$10 per month per project. You may use the `incrementQuantity` and `decrementQuantity` methods to easily increment or decrement your subscription quantity:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('default')->incrementQuantity(5);

$user->subscription('default')->decrementQuantity();

// Subtract five from the subscription's current quantity...
$user->subscription('default')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('default')->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without prorating the charges:

```
$user->subscription('default')->noProrate()->updateQuantity(10);
```

For more information on subscription quantities, consult the [Stripe documentation](#).

Quantities for Subscriptions With Multiple Products

If your subscription is a [subscription with multiple products](#), you should pass the ID of the price whose quantity you wish to increment or decrement as the second argument to the increment / decrement methods:

```
$user->subscription('default')->incrementQuantity(1, 'price_chat');
```

Subscriptions With Multiple Products

[Subscription with multiple products](#) allow you to assign multiple billing products to a single subscription. For example, imagine you are building a customer service "helpdesk" application that has a base subscription price of \$10 per month but offers a live chat add-on product for an additional \$15 per month. Information for subscriptions with multiple products is stored in Cashier's `subscription_items` database table.

You may specify multiple products for a given subscription by passing an array of prices as the second argument to the `newSubscription` method:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default', [
        'price_monthly',
        'price_chat',
    ])->create($request->paymentMethodId);

    // ...
});
```

In the example above, the customer will have two prices attached to their `default` subscription. Both prices will be charged on their respective billing intervals. If necessary, you may use the `quantity` method to indicate a specific quantity for each price:

```
$user = User::find(1);

$user->newSubscription('default', ['price_monthly', 'price_chat'])
    ->quantity(5, 'price_chat')
    ->create($paymentMethod);
```

If you would like to add another price to an existing subscription, you may invoke the subscription's `addPrice` method:

```
$user = User::find(1);

$user->subscription('default')->addPrice('price_chat');
```

The example above will add the new price and the customer will be billed for it on their next billing cycle. If you would like to bill the customer immediately you may use the `addPriceAndInvoice` method:

```
$user->subscription('default')->addPriceAndInvoice('price_chat');
```

If you would like to add a price with a specific quantity, you can pass the quantity as the second argument of the `addPrice` or `addPriceAndInvoice` methods:

```
$user = User::find(1);

$user->subscription('default')->addPrice('price_chat', 5);
```

You may remove prices from subscriptions using the `removePrice` method:

```
$user->subscription('default')->removePrice('price_chat');
```



You may not remove the last price on a subscription. Instead, you should simply cancel the subscription.

Swapping Prices

You may also change the prices attached to a subscription with multiple products. For example, imagine a customer has a `price_basic` subscription with a `price_chat` add-on product and you want to upgrade the customer from the `price_basic` to the `price_pro` price:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->swap(['price_pro', 'price_chat']);
```

When executing the example above, the underlying subscription item with the `price_basic` is deleted and the one with the `price_chat` is preserved. Additionally, a new subscription item for the `price_pro` is created.

You can also specify subscription item options by passing an array of key / value pairs to the `swap` method. For example, you may need to specify the subscription price quantities:

```
$user = User::find(1);

$user->subscription('default')->swap([
    'price_pro' => ['quantity' => 5],
    'price_chat'
]);
```

If you want to swap a single price on a subscription, you may do so using the `swap` method on the subscription item itself. This approach is particularly useful if you would like to preserve all of the existing metadata on the subscription's other prices:

```
$user = User::find(1);

$user->subscription('default')
    ->findItemOrFail('price_basic')
    ->swap('price_pro');
```

Proration

By default, Stripe will prorate charges when adding or removing prices from a subscription with multiple products. If you would like to make a price adjustment without proration, you should chain the `noProrate` method onto your price operation:

```
$user->subscription('default')->noProrate()->removePrice('price_chat');
```

Quantities

If you would like to update quantities on individual subscription prices, you may do so using the [existing quantity methods](#) by passing the ID of the price as an additional argument to the method:

```
$user = User::find(1);

$user->subscription('default')->incrementQuantity(5, 'price_chat');

$user->subscription('default')->decrementQuantity(3, 'price_chat');

$user->subscription('default')->updateQuantity(10, 'price_chat');
```



When a subscription has multiple prices the `stripe_price` and `quantity` attributes on the `Subscription` model will be `null`. To access the individual price attributes, you should use the `items` relationship available on the `Subscription` model.

Subscription Items

When a subscription has multiple prices, it will have multiple subscription "items" stored in your database's `subscription_items` table. You may access these via the `items` relationship on the subscription:

```
use App\Models\User;

$user = User::find(1);

$subscriptionItem = $user->subscription('default')->items->first();

// Retrieve the Stripe price and quantity for a specific item...
$stripePrice = $subscriptionItem->stripe_price;
$quantity = $subscriptionItem->quantity;
```

You can also retrieve a specific price using the `findItemOrFail` method:

```
$user = User::find(1);

$subscriptionItem = $user->subscription('default')->findItemOrFail('price_chat');
```

Multiple Subscriptions

Stripe allows your customers to have multiple subscriptions simultaneously. For example, you may run a gym that offers a swimming subscription and a weight-lifting subscription, and each subscription may have different pricing. Of course, customers should be able to subscribe to either or both plans.

When your application creates subscriptions, you may provide the type of the subscription to the `newSubscription` method. The type may be any string that represents the type of subscription the user is initiating:

```
use Illuminate\Http\Request;

Route::post('/swimming/subscribe', function (Request $request) {
    $request->user()->newSubscription('swimming')
        ->price('price_swimming_monthly')
        ->create($request->paymentMethodId);

    // ...
});
```

In this example, we initiated a monthly swimming subscription for the customer. However, they may want to swap to a yearly subscription at a later time. When adjusting the customer's subscription, we can simply swap the price on the `swimming` subscription:

```
$user->subscription('swimming')->swap('price_swimming_yearly');
```

Of course, you may also cancel the subscription entirely:

```
$user->subscription('swimming')->cancel();
```

Metered Billing

Metered billing allows you to charge customers based on their product usage during a billing cycle. For example, you may charge customers based on the number of text messages or emails they send per month.

To start using metered billing, you will first need to create a new product in your Stripe dashboard with a metered price. Then, use the `meteredPrice` to add the metered price ID to a customer subscription:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default')
        ->meteredPrice('price_metered')
        ->create($request->paymentMethodId);

    // ...
});
```

You may also start a metered subscription via [Stripe Checkout](#):

```
$checkout = Auth::user()
    ->newSubscription('default', [])
    ->meteredPrice('price_metered')
    ->checkout();

return view('your-checkout-view', [
    'checkout' => $checkout,
]);
```

Reporting Usage

As your customer uses your application, you will report their usage to Stripe so that they can be billed accurately. To increment the usage of a metered subscription, you may use the `reportUsage` method:

```
$user = User::find(1);

$user->subscription('default')->reportUsage();
```

By default, a "usage quantity" of 1 is added to the billing period. Alternatively, you may pass a specific amount of "usage" to add to the customer's usage for the billing period:

```
$user = User::find(1);

$user->subscription('default')->reportUsage(15);
```

If your application offers multiple prices on a single subscription, you will need to use the `reportUsageFor` method to specify the metered price you want to report usage for:

```
$user = User::find(1);

$user->subscription('default')->reportUsageFor('price_metered', 15);
```

Sometimes, you may need to update usage which you have previously reported. To accomplish this, you may pass a timestamp or a `DateTimeInterface` instance as the second parameter to `reportUsage`. When doing so, Stripe will update the usage that was reported at that given time. You can continue to update previous usage records as the given date and time is still within the current billing period:

```
$user = User::find(1);

$user->subscription('default')->reportUsage(5, $timestamp);
```

Retrieving Usage Records

To retrieve a customer's past usage, you may use a subscription instance's `usageRecords` method:

```
$user = User::find(1);

$usageRecords = $user->subscription('default')->usageRecords();
```

If your application offers multiple prices on a single subscription, you may use the `usageRecordsFor` method to specify the metered price that you wish to retrieve usage records for:

```
$user = User::find(1);

$usageRecords = $user->subscription('default')->usageRecordsFor('price_metered');
```

The `usageRecords` and `usageRecordsFor` methods return a Collection instance containing an associative array of usage records. You may iterate over this array to display a customer's total usage:

```
@foreach ($usageRecords as $usageRecord)
    - Period Starting: {{ $usageRecord['period']['start'] }}
    - Period Ending: {{ $usageRecord['period']['end'] }}
    - Total Usage: {{ $usageRecord['total_usage'] }}
@endforeach
```

For a full reference of all usage data returned and how to use Stripe's cursor based pagination, please consult [the official Stripe API documentation](#).

Subscription Taxes



Instead of calculating Tax Rates manually, you can automatically calculate taxes using Stripe Tax

To specify the tax rates a user pays on a subscription, you should implement the `taxRates` method on your billable model and return an array containing the Stripe tax rate IDs. You can define these tax rates in [your Stripe dashboard](#):

```
/**
 * The tax rates that should apply to the customer's subscriptions.
 *
 * @return array<int, string>
 */
public function taxRates(): array
{
    return ['txr_id'];
}
```

The `taxRates` method enables you to apply a tax rate on a customer-by-customer basis, which may be helpful for a user base that spans multiple countries and tax rates.

If you're offering subscriptions with multiple products, you may define different tax rates for each price by implementing a `priceTaxRates` method on your billable model:

```
/**
 * The tax rates that should apply to the customer's subscriptions.
 *
 * @return array<string, array<int, string>>
 */
public function priceTaxRates(): array
{
    return [
        'price_monthly' => ['txr_id'],
    ];
}
```



The `taxRates` method only applies to subscription charges. If you use Cashier to make "one-off" charges, you will need to manually specify the tax rate at that time.

Syncing Tax Rates

When changing the hard-coded tax rate IDs returned by the `taxRates` method, the tax settings on any existing subscriptions for the user will remain the same. If you wish to update the tax value for existing subscriptions with the new `taxRates` values, you should call the `syncTaxRates` method on the user's subscription instance:

```
$user->subscription('default')->syncTaxRates();
```

This will also sync any item tax rates for a subscription with multiple products. If your application is offering subscriptions with multiple products, you should ensure that your billable model implements the `priceTaxRates` method [discussed above](#).

Tax Exemption

Cashier also offers the `isNotTaxExempt`, `isTaxExempt`, and `reverseChargeApplies` methods to determine if the customer is tax exempt. These methods will call the Stripe API to determine a customer's tax exemption status:

```
use App\Models\User;

$user = User::find(1);

$user->isTaxExempt();
$user->isNotTaxExempt();
$user->reverseChargeApplies();
```



These methods are also available on any `Laravel\Cashier\Invoice` object. However, when invoked on an `Invoice` object, the methods will determine the exemption status at the time the invoice was created.

Subscription Anchor Date

By default, the billing cycle anchor is the date the subscription was created or, if a trial period is used, the date that the trial ends. If you would like to modify the billing anchor date, you may use the `anchorBillingCycleOn` method:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $anchor = Carbon::parse('first day of next month');

    $request->user()->newSubscription('default', 'price_monthly')
        ->anchorBillingCycleOn($anchor->startOfDay())
        ->create($request->paymentMethodId);

    // ...
});
```

For more information on managing subscription billing cycles, consult the [Stripe billing cycle documentation](#).

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription('default')->cancel();
```

When a subscription is canceled, Cashier will automatically set the `ends_at` column in your `subscriptions` database table. This column is used to know when the `subscribed` method should begin returning `false`.

For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has canceled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('default')->onGracePeriod()) {
    // ...
}
```

If you wish to cancel a subscription immediately, call the `cancelNow` method on the user's subscription:

```
$user->subscription('default')->cancelNow();
```

If you wish to cancel a subscription immediately and invoice any remaining un-invoiced metered usage or new / pending proration invoice items, call the `cancelNowAndInvoice` method on the user's subscription:

```
$user->subscription('default')->cancelNowAndInvoice();
```

You may also choose to cancel the subscription at a specific moment in time:

```
$user->subscription('default')->cancelAt(
    now()->addDays(10)
);
```

Finally, you should always cancel user subscriptions before deleting the associated user model:

```
$user->subscription('default')->cancelNow();

$user->delete();
```

Resuming Subscriptions

If a customer has canceled their subscription and you wish to resume it, you may invoke the `resume` method on the subscription. The customer must still be within their "grace period" in order to resume a subscription:

```
$user->subscription('default')->resume();
```

If the customer cancels a subscription and then resumes that subscription before the subscription has fully expired the customer will not be billed immediately. Instead, their subscription will be re-activated and they will be billed on the original billing cycle.

Subscription Trials

With Payment Method Up Front

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should use the `trialDays` method when creating your subscriptions:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default', 'price_monthly')
        ->trialDays(10)
        ->create($request->paymentMethodId);

    // ...
});
```

This method will set the trial period ending date on the subscription record within the database and instruct Stripe to not begin billing the customer until after this date. When using the `trialDays` method, Cashier will overwrite any default trial period configured for the price in Stripe.



If the customer's subscription is not canceled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

The `trialUntil` method allows you to provide a `DateTime` instance that specifies when the trial period should end:

```
use Carbon\Carbon;

$user->newSubscription('default', 'price_monthly')
    ->trialUntil(Carbon::now()->addDays(10))
    ->create($paymentMethod);
```

You may determine if a user is within their trial period using either the `onTrial` method of the user instance or the `onTrial` method of the subscription instance. The two examples below are equivalent:

```
if ($user->onTrial('default')) {
    // ...
}

if ($user->subscription('default')->onTrial()) {
    // ...
}
```

You may use the `endTrial` method to immediately end a subscription trial:

```
$user->subscription('default')->endTrial();
```

To determine if an existing trial has expired, you may use the `hasExpiredTrial` methods:

```
if ($user->hasExpiredTrial('default')) {
    // ...
}

if ($user->subscription('default')->hasExpiredTrial()) {
    // ...
}
```

Defining Trial Days in Stripe / Cashier

You may choose to define how many trial days your price's receive in the Stripe dashboard or always pass them explicitly using Cashier. If you choose to define your price's trial days in Stripe you should be aware that new subscriptions, including new subscriptions for a customer that had a subscription in the past, will always receive a trial period unless you explicitly call the `skipTrial()` method.

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the user record to your desired trial ending date. This is typically done during user registration:

```
use App\Models\User;

$user = User::create([
    // ...
    'trial_ends_at' => now()->addDays(10),
]);
```



Be sure to add a `date cast` for the `trial_ends_at` attribute within your billable model's class definition.

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the billable model instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
$user = User::find(1);

$user->newSubscription('default', 'price_monthly')->create($paymentMethod);
```

To retrieve the user's trial ending date, you may use the `trialEndsAt` method. This method will return a Carbon date instance if a user is on a trial or `null` if they aren't. You may also pass an optional subscription type parameter if you would like to get the trial ending date for a specific subscription other than the default one:

```
if ($user->onTrial()) {
    $trialEndsAt = $user->trialEndsAt('main');
}
```

You may also use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not yet created an actual subscription:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

Extending Trials

The `extendTrial` method allows you to extend the trial period of a subscription after the subscription has been created. If the trial has already expired and the customer is already being billed for the subscription, you can still offer them an extended trial. The time spent within the trial period will be deducted from the customer's next invoice:

```
use App\Models\User;

$subscription = User::find(1)->subscription('default');

// End the trial 7 days from now...
$subscription->extendTrial(
    now()->addDays(7)
);

// Add an additional 5 days to the trial...
$subscription->extendTrial(
    $subscription->trial_ends_at->addDays(5)
);
```

Handling Stripe Webhooks



You may use [the Stripe CLI](#) to help test webhooks during local development.

Stripe can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is automatically registered by the Cashier service provider. This controller will handle all incoming webhook requests.

By default, the Cashier webhook controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Stripe settings), customer updates, customer deletions, subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any Stripe webhook event you like.

To ensure your application can handle Stripe webhooks, be sure to configure the webhook URL in the Stripe control panel. By default, Cashier's webhook controller responds to the `/stripe/webhook` URL path. The full list of all webhooks you should enable in the Stripe control panel are:

- `customer.subscription.created`
- `customer.subscription.updated`
- `customer.subscription.deleted`
- `customer.updated`
- `customer.deleted`
- `payment_method.automatic_update`
- `invoice.payment_action_required`
- `invoice.payment_succeeded`

For convenience, Cashier includes a `cashier:webhook` Artisan command. This command will create a webhook in Stripe that listens to all of the events required by Cashier:

```
php artisan cashier:webhook
```

By default, the created webhook will point to the URL defined by the `APP_URL` environment variable and the `cashier.webhook` route that is included with Cashier. You may provide the `--url` option when invoking the command if you would like to use a different URL:

```
php artisan cashier:webhook --url "https://example.com/stripe/webhook"
```

The webhook that is created will use the Stripe API version that your version of Cashier is compatible with. If you would like to use a different Stripe version, you may provide the `--api-version` option:

```
php artisan cashier:webhook --api-version="2019-12-03"
```

After creation, the webhook will be immediately active. If you wish to create the webhook but have it disabled until you're ready, you may provide the `--disabled` option when invoking the command:

```
php artisan cashier:webhook --disabled
```



Make sure you protect incoming Stripe webhook requests with Cashier's included [webhook signature verification middleware](#).

Webhooks and CSRF Protection

Since Stripe webhooks need to bypass Laravel's [CSRF protection](#), be sure to list the URI as an exception in your application's `App\Http\Middleware\VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'stripe/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellations for failed charges and other common Stripe webhook events. However, if you have additional webhook events you would like to handle, you may do so by listening to the following events that are dispatched by Cashier:

- `Laravel\Cashier\Events\WebhookReceived`
- `Laravel\Cashier\Events\WebhookHandled`

Both events contain the full payload of the Stripe webhook. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you may register a [listener](#) that will handle the event:

```
<?php

namespace App\Listeners;

use Laravel\Cashier\Events\WebhookReceived;

class StripeEventListener
{
    /**
     * Handle received Stripe webhooks.
     */
    public function handle(WebhookReceived $event): void
    {
        if ($event->payload['type'] === 'invoice.payment_succeeded') {
            // Handle the incoming event...
        }
    }
}
```

Once your listener has been defined, you may register it within your application's `EventServiceProvider`:

```
<?php

namespace App\Providers;

use App\Listeners\StripeEventListener;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Laravel\Cashier\Events\WebhookReceived;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        WebhookReceived::class => [
            StripeEventListener::class,
        ],
    ];
}
```

Verifying Webhook Signatures

To secure your webhooks, you may use [Stripe's webhook signatures](#). For convenience, Cashier automatically includes a middleware which validates that the incoming Stripe webhook request is valid.

To enable webhook verification, ensure that the `STRIPE_WEBHOOK_SECRET` environment variable is set in your application's `.env` file. The webhook `secret` may be retrieved from your Stripe account dashboard.

Single Charges

Simple Charge

If you would like to make a one-time charge against a customer, you may use the `charge` method on a billable model instance. You will need to [provide a payment method identifier](#) as the second argument to the `charge` method:

```
use Illuminate\Http\Request;

Route::post('/purchase', function (Request $request) {
    $stripeCharge = $request->user()->charge(
        100, $request->paymentMethodId
    );

    // ...
});
```

The `charge` method accepts an array as its third argument, allowing you to pass any options you wish to the underlying Stripe charge creation. More information regarding the options available to you when creating charges may be found in the [Stripe documentation](#):

```
$user->charge(100, $paymentMethod, [
    'custom_option' => $value,
]);
```

You may also use the `charge` method without an underlying customer or user. To accomplish this, invoke the `charge` method on a new instance of your application's billable model:

```
use App\Models\User;

$stripeCharge = (new User)->charge(100, $paymentMethod);
```

The `charge` method will throw an exception if the charge fails. If the charge is successful, an instance of `Laravel\Cashier\Payment` will be returned from the method:

```
try {
    $payment = $user->charge(100, $paymentMethod);
} catch (Exception $e) {
    // ...
}
```



The `charge` method accepts the payment amount in the lowest denominator of the currency used by your application. For example, if customers are paying in United States Dollars, amounts should be specified in pennies.

Charge With Invoice

Sometimes you may need to make a one-time charge and offer a PDF invoice to your customer. The `invoicePrice` method lets you do just that. For example, let's invoice a customer for five new shirts:

```
$user->invoicePrice('price_tshirt', 5);
```

The invoice will be immediately charged against the user's default payment method. The `invoicePrice` method also accepts an array as its third argument. This array contains the billing options for the invoice item. The fourth argument accepted by the method is also an array which should contain the billing options for the invoice itself:

```
$user->invoicePrice('price_tshirt', 5, [
    'discounts' => [
        ['coupon' => 'SUMMER21SALE']
    ],
], [
    'default_tax_rates' => ['txr_id'],
]);
});
```

Similarly to `invoicePrice`, you may use the `tabPrice` method to create a one-time charge for multiple items (up to 250 items per invoice) by adding them to the customer's "tab" and then invoicing the customer. For example, we may invoice a customer for five shirts and two mugs:

```
$user->tabPrice('price_tshirt', 5);
$user->tabPrice('price_mug', 2);
$user->invoice();
```

Alternatively, you may use the `invoiceFor` method to make a "one-off" charge against the customer's default payment method:

```
$user->invoiceFor('One Time Fee', 500);
```

Although the `invoiceFor` method is available for you to use, it is recommended that you use the `invoicePrice` and `tabPrice` methods with pre-defined prices. By doing so, you will have access to better analytics and data within your Stripe dashboard regarding your sales on a per-product basis.



The `invoice`, `invoicePrice`, and `invoiceFor` methods will create a Stripe invoice which will retry failed billing attempts. If you do not want invoices to retry failed charges, you will need to close them using the Stripe API after the first failed charge.

Creating Payment Intents

You can create a new Stripe payment intent by invoking the `pay` method on a billable model instance. Calling this method will create a payment intent that is wrapped in a `Laravel\Cashier\Payment` instance:

```
use Illuminate\Http\Request;

Route::post('/pay', function (Request $request) {
    $payment = $request->user()->pay(
        $request->get('amount')
    );

    return $payment->client_secret;
});
```

After creating the payment intent, you can return the client secret to your application's frontend so that the user can complete the payment in their browser. To read more about building entire payment flows using Stripe payment intents, please consult the [Stripe documentation](#).

When using the `pay` method, the default payment methods that are enabled within your Stripe dashboard will be available to the customer. Alternatively, if you only want to allow for some specific payment methods to be used, you may use the `payWith` method:

```
use Illuminate\Http\Request;

Route::post('/pay', function (Request $request) {
    $payment = $request->user()->payWith(
        $request->get('amount'), ['card', 'bancontact']
    );

    return $payment->client_secret;
});
```



The `pay` and `payWith` methods accept the payment amount in the lowest denominator of the currency used by your application. For example, if customers are paying in United States Dollars, amounts should be specified in pennies.

Refunding Charges

If you need to refund a Stripe charge, you may use the `refund` method. This method accepts the Stripe [payment intent ID](#) as its first argument:

```
$payment = $user->charge(100, $paymentMethodId);

$user->refund($payment->id);
```

Invoices

Retrieving Invoices

You may easily retrieve an array of a billable model's invoices using the `invoices` method. The `invoices` method returns a collection of `Laravel\Cashier\Invoice` instances:

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

If you would like to include pending invoices in the results, you may use the `invoicesIncludingPending` method:

```
$invoices = $user->invoicesIncludingPending();
```

You may use the `findInvoice` method to retrieve a specific invoice by its ID:

```
$invoice = $user->findInvoice($invoiceId);
```

Displaying Invoice Information

When listing the invoices for the customer, you may use the invoice's methods to display the relevant invoice information. For example, you may wish to list every invoice in a table, allowing the user to easily download any of them:

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>
```

Upcoming Invoices

To retrieve the upcoming invoice for a customer, you may use the `upcomingInvoice` method:

```
$invoice = $user->upcomingInvoice();
```

Similarly, if the customer has multiple subscriptions, you can also retrieve the upcoming invoice for a specific subscription:

```
$invoice = $user->subscription('default')->upcomingInvoice();
```

Previewing Subscription Invoices

Using the `previewInvoice` method, you can preview an invoice before making price changes. This will allow you to determine what your customer's invoice will look like when a given price change is made:

```
$invoice = $user->subscription('default')->previewInvoice('price_yearly');
```

You may pass an array of prices to the `previewInvoice` method in order to preview invoices with multiple new prices:

```
$invoice = $user->subscription('default')->previewInvoice(['price_yearly', 'price_metered']);
```

Generating Invoice PDFs

Before generating invoice PDFs, you should use Composer to install the Dompdf library, which is the default invoice renderer for Cashier:

```
composer require dompdf/dompdf
```

From within a route or controller, you may use the `downloadInvoice` method to generate a PDF download of a given invoice. This method will automatically generate the proper HTTP response needed to download the invoice:

```
use Illuminate\Http\Request;

Route::get('/user/invoice/{invoice}', function (Request $request, string $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId);
});
```

By default, all data on the invoice is derived from the customer and invoice data stored in Stripe. The filename is based on your `app.name` config value. However, you can customize some of this data by providing an array as the second argument to the `downloadInvoice` method. This array allows you to customize information such as your company and product details:

```
return $request->user()->downloadInvoice($invoiceId, [
    'vendor' => 'Your Company',
    'product' => 'Your Product',
    'street' => 'Main Str. 1',
    'location' => '2000 Antwerp, Belgium',
    'phone' => '+32 499 00 00 00',
    'email' => 'info@example.com',
    'url' => 'https://example.com',
    'vendorVat' => 'BE123456789',
]);

```

The `downloadInvoice` method also allows for a custom filename via its third argument. This filename will automatically be suffixed with `.pdf`:

```
return $request->user()->downloadInvoice($invoiceId, [], 'my-invoice');
```

Custom Invoice Renderer

Cashier also makes it possible to use a custom invoice renderer. By default, Cashier uses the `DompdfInvoiceRenderer` implementation, which utilizes the `dompdf` PHP library to generate Cashier's invoices. However, you may use any renderer you wish by implementing the `Laravel\Cashier\Contracts\InvoiceRenderer` interface. For example, you may wish to render an invoice PDF using an API call to a third-party PDF rendering service:

```
use Illuminate\Support\Facades\Http;
use Laravel\Cashier\Contracts\InvoiceRenderer;
use Laravel\Cashier\Invoice;

class ApiInvoiceRenderer implements InvoiceRenderer
{
    /**
     * Render the given invoice and return the raw PDF bytes.
     */
    public function render(Invoice $invoice, array $data = [], array $options = []): string
    {
        $html = $invoice->view($data)->render();

        return Http::get('https://example.com/html-to-pdf', ['html' => $html])->get()->body();
    }
}
```

Once you have implemented the invoice renderer contract, you should update the `cashier.invoices.renderer` configuration value in your application's `config/cashier.php` configuration file. This configuration value should be set to the class name of your custom renderer implementation.

Checkout

Cashier Stripe also provides support for [Stripe Checkout](#). Stripe Checkout takes the pain out of implementing custom pages to accept payments by providing a pre-built, hosted payment page.

The following documentation contains information on how to get started using Stripe Checkout with Cashier. To learn more about Stripe Checkout, you should also consider reviewing [Stripe's own documentation on Checkout](#).

Product Checkouts

You may perform a checkout for an existing product that has been created within your Stripe dashboard using the `checkout` method on a billable model. The `checkout` method will initiate a new Stripe Checkout session. By default, you're required to pass a Stripe Price ID:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout('price_tshirt');
});
```

If needed, you may also specify a product quantity:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 15]);
});
```

When a customer visits this route they will be redirected to Stripe's Checkout page. By default, when a user successfully completes or cancels a purchase they will be redirected to your `home` route location, but you may specify custom callback URLs using the `success_url` and `cancel_url` options:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 1], [
        'success_url' => route('your-success-route'),
        'cancel_url' => route('your-cancel-route'),
    ]);
});
```

When defining your `success_url` checkout option, you may instruct Stripe to add the checkout session ID as a query string parameter when invoking your URL. To do so, add the literal string `{CHECKOUT_SESSION_ID}` to your `success_url` query string. Stripe will replace this placeholder with the actual checkout session ID:

```
use Illuminate\Http\Request;
use Stripe\Checkout\Session;
use Stripe\Customer;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 1], [
        'success_url' => route('checkout-success').'?session_id={CHECKOUT_SESSION_ID}',
        'cancel_url' => route('checkout-cancel'),
    ]);
});

Route::get('/checkout-success', function (Request $request) {
    $checkoutSession = $request->user()->stripe()->checkout->sessions->retrieve($request->get('session_id'));

    return view('checkout.success', ['checkoutSession' => $checkoutSession]);
})->name('checkout-success');
```

Promotion Codes

By default, Stripe Checkout does not allow `user_redeemable_promotion_codes`. Luckily, there's an easy way to enable these for your Checkout page. To do so, you may invoke the `allowPromotionCodes` method:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()
        ->allowPromotionCodes()
        ->checkout('price_tshirt');
});
```

Single Charge Checkouts

You can also perform a simple charge for an ad-hoc product that has not been created in your Stripe dashboard. To do so you may use the `checkoutCharge` method on a billable model and pass it a chargeable amount, a product name, and an optional quantity. When a customer visits this route they will be redirected to Stripe's Checkout page:

```
use Illuminate\Http\Request;

Route::get('/charge-checkout', function (Request $request) {
    return $request->user()->checkoutCharge(1200, 'T-Shirt', 5);
});
```



When using the `checkoutCharge` method, Stripe will always create a new product and price in your Stripe dashboard. Therefore, we recommend that you create the products up front in your Stripe dashboard and use the `checkout` method instead.

Subscription Checkouts



Using Stripe Checkout for subscriptions requires you to enable the `customer.subscription.created` webhook in your Stripe dashboard. This webhook will create the subscription record in your database and store all of the relevant subscription items.

You may also use Stripe Checkout to initiate subscriptions. After defining your subscription with Cashier's subscription builder methods, you may call the `checkout` method. When a customer visits this route they will be redirected to Stripe's Checkout page:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->checkout();
});
```

Just as with product checkouts, you may customize the success and cancellation URLs:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->checkout([
            'success_url' => route('your-success-route'),
            'cancel_url' => route('your-cancel-route'),
        ]);
});
```

Of course, you can also enable promotion codes for subscription checkouts:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->allowPromotionCodes()
        ->checkout();
});
```

 Unfortunately Stripe Checkout does not support all subscription billing options when starting subscriptions. Using the `anchorBillingCycleOn` method on the subscription builder, setting proration behavior, or setting payment behavior will not have any effect during Stripe Checkout sessions. Please consult [the Stripe Checkout Session API documentation](#) to review which parameters are available.

Stripe Checkout and Trial Periods

Of course, you can define a trial period when building a subscription that will be completed using Stripe Checkout:

```
$checkout = Auth::user()->newSubscription('default', 'price_monthly')
->trialDays(3)
->checkout();
```

However, the trial period must be at least 48 hours, which is the minimum amount of trial time supported by Stripe Checkout.

Subscriptions and Webhooks

Remember, Stripe and Cashier update subscription statuses via webhooks, so there's a possibility a subscription might not yet be active when the customer returns to the application after entering their payment information. To handle this scenario, you may wish to display a message informing the user that their payment or subscription is pending.

Collecting Tax IDs

Checkout also supports collecting a customer's Tax ID. To enable this on a checkout session, invoke the `collectTaxIds` method when creating the session:

```
$checkout = $user->collectTaxIds()->checkout('price_tshirt');
```

When this method is invoked, a new checkbox will be available to the customer that allows them to indicate if they're purchasing as a company. If so, they will have the opportunity to provide their Tax ID number.



If you have already configured [automatic tax collection](#) in your application's service provider then this feature will be enabled automatically and there is no need to invoke the `collectTaxIds` method.

Guest Checkouts

Using the `Checkout::guest` method, you may initiate checkout sessions for guests of your application that do not have an "account":

```
use Illuminate\Http\Request;
use Laravel\Cashier\Checkout;

Route::get('/product-checkout', function (Request $request) {
    return Checkout::guest()->create('price_tshirt', [
        'success_url' => route('your-success-route'),
        'cancel_url' => route('your-cancel-route'),
    ]);
});
```

Similarly to when creating checkout sessions for existing users, you may utilize additional methods available on the `Laravel\Cashier\CheckoutBuilder` instance to customize the guest checkout session:

```
use Illuminate\Http\Request;
use Laravel\Cashier\Checkout;

Route::get('/product-checkout', function (Request $request) {
    return Checkout::guest()
        ->withPromotionCode('promo-code')
        ->create('price_tshirt', [
            'success_url' => route('your-success-route'),
            'cancel_url' => route('your-cancel-route'),
        ]);
});
```

After a guest checkout has been completed, Stripe can dispatch a `checkout.session.completed` webhook event, so make sure to [configure your Stripe webhook](#) to actually send this event to your application. Once the webhook has been enabled within the Stripe dashboard, you may [handle the webhook with Cashier](#). The object contained in the webhook payload will be a `checkout` object that you may inspect in order to fulfill your customer's order.

Handling Failed Payments

Sometimes, payments for subscriptions or single charges can fail. When this happens, Cashier will throw an `Laravel\Cashier\Exceptions\IncompletePayment` exception that informs you that this happened. After catching this exception, you have two options on how to proceed.

First, you could redirect your customer to the dedicated payment confirmation page which is included with Cashier. This page already has an associated named route that is registered via Cashier's service provider. So, you may catch the `IncompletePayment` exception and redirect the user to the payment confirmation page:

```
use Laravel\Cashier\Exceptions\IncompletePayment;

try {
    $subscription = $user->newSubscription('default', 'price_monthly')
        ->create($paymentMethod);
} catch (IncompletePayment $exception) {
    return redirect()->route(
        'cashier.payment',
        [$exception->payment->id, 'redirect' => route('home')]
    );
}
```

On the payment confirmation page, the customer will be prompted to enter their credit card information again and perform any additional actions required by Stripe, such as "3D Secure" confirmation. After confirming their payment, the user will be redirected to the URL provided by the `redirect` parameter specified above. Upon redirection, `message` (string) and `success` (integer) query string variables will be added to the URL. The payment page currently supports the following payment method types:

- Credit Cards
- Alipay
- Bancontact
- BECS Direct Debit
- EPS
- Giropay
- iDEAL
- SEPA Direct Debit

Alternatively, you could allow Stripe to handle the payment confirmation for you. In this case, instead of redirecting to the payment confirmation page, you may [setup Stripe's automatic billing emails](#) in your Stripe dashboard. However, if an `IncompletePayment` exception is caught, you should still inform the user they will receive an email with further payment confirmation instructions.

Payment exceptions may be thrown for the following methods: `charge`, `invoiceFor`, and `invoice` on models using the `Billable` trait. When interacting with subscriptions, the `create` method on the `SubscriptionBuilder`, and the `incrementAndInvoice` and `swapAndInvoice` methods on the `Subscription` and `SubscriptionItem` models may throw incomplete payment exceptions.

Determining if an existing subscription has an incomplete payment may be accomplished using the `hasIncompletePayment` method on the billable model or a subscription instance:

```
if ($user->hasIncompletePayment('default')) {
    // ...
}

if ($user->subscription('default')->hasIncompletePayment()) {
    // ...
}
```

You can derive the specific status of an incomplete payment by inspecting the `payment` property on the exception instance:

```
use Laravel\Cashier\Exceptions\IncompletePayment;

try {
    $user->charge(1000, 'pm_card_threeDSecure2Required');
} catch (IncompletePayment $exception) {
    // Get the payment intent status...
    $exception->payment->status;

    // Check specific conditions...
    if ($exception->payment->requiresPaymentMethod()) {
        // ...
    } elseif ($exception->payment->requiresConfirmation()) {
        // ...
    }
}
```

Confirming Payments

Some payment methods require additional data in order to confirm payments. For example, SEPA payment methods require additional "mandate" data during the payment process. You may provide this data to Cashier using the

`withPaymentConfirmationOptions` method:

```
$subscription->withPaymentConfirmationOptions([
    'mandate_data' => '...',
])->swap('price_xxx');
```

You may consult the [Stripe API documentation](#) to review all of the options accepted when confirming payments.

Strong Customer Authentication

If your business or one of your customers is based in Europe you will need to abide by the EU's Strong Customer Authentication (SCA) regulations. These regulations were imposed in September 2019 by the European Union to prevent payment fraud. Luckily, Stripe and Cashier are prepared for building SCA compliant applications.



Before getting started, review [Stripe's guide on PSD2 and SCA](#) as well as their [documentation on the new SCA APIs](#).

Payments Requiring Additional Confirmation

SCA regulations often require extra verification in order to confirm and process a payment. When this happens, Cashier will throw a `Laravel\Cashier\Exceptions\IncompletePayment` exception that informs you that extra verification is needed. More information on how to handle these exceptions can be found in the documentation on [handling failed payments](#).

Payment confirmation screens presented by Stripe or Cashier may be tailored to a specific bank or card issuer's payment flow and can include additional card confirmation, a temporary small charge, separate device authentication, or other forms of verification.

Incomplete and Past Due State

When a payment needs additional confirmation, the subscription will remain in an `incomplete` or `past_due` state as indicated by its `stripe_status` database column. Cashier will automatically activate the customer's subscription as

soon as payment confirmation is complete and your application is notified by Stripe via webhook of its completion.

For more information on `incomplete` and `past_due` states, please refer to [our additional documentation on these states](#).

Off-Session Payment Notifications

Since SCA regulations require customers to occasionally verify their payment details even while their subscription is active, Cashier can send a notification to the customer when off-session payment confirmation is required. For example, this may occur when a subscription is renewing. Cashier's payment notification can be enabled by setting the `CASHIER_PAYMENT_NOTIFICATION` environment variable to a notification class. By default, this notification is disabled. Of course, Cashier includes a notification class you may use for this purpose, but you are free to provide your own notification class if desired:

```
CASHIER_PAYMENT_NOTIFICATION=Laravel\Cashier\Notifications\ConfirmPayment
```

To ensure that off-session payment confirmation notifications are delivered, verify that [Stripe webhooks are configured](#) for your application and the `invoice.payment_action_required` webhook is enabled in your Stripe dashboard. In addition, your `Billable` model should also use Laravel's `Illuminate\Notifications\Notifiable` trait.



Notifications will be sent even when customers are manually making a payment that requires additional confirmation. Unfortunately, there is no way for Stripe to know that the payment was done manually or "off-session". But, a customer will simply see a "Payment Successful" message if they visit the payment page after already confirming their payment. The customer will not be allowed to accidentally confirm the same payment twice and incur an accidental second charge.

Stripe SDK

Many of Cashier's objects are wrappers around Stripe SDK objects. If you would like to interact with the Stripe objects directly, you may conveniently retrieve them using the `asStripe` method:

```
$stripeSubscription = $subscription->asStripeSubscription();
$stripeSubscription->application_fee_percent = 5;
$stripeSubscription->save();
```

You may also use the `updateStripeSubscription` method to update a Stripe subscription directly:

```
$subscription->updateStripeSubscription(['application_fee_percent' => 5]);
```

You may invoke the `stripe` method on the `Cashier` class if you would like to use the `Stripe\StripeClient` client directly. For example, you could use this method to access the `StripeClient` instance and retrieve a list of prices from your Stripe account:

```
use Laravel\Cashier\Cashier;
$prices = Cashier::stripe()->prices->all();
```

Testing

When testing an application that uses Cashier, you may mock the actual HTTP requests to the Stripe API; however, this requires you to partially re-implement Cashier's own behavior. Therefore, we recommend allowing your tests to hit the actual Stripe API. While this is slower, it provides more confidence that your application is working as expected and any slow tests may be placed within their own PHPUnit testing group.

When testing, remember that Cashier itself already has a great test suite, so you should only focus on testing the subscription and payment flow of your own application and not every underlying Cashier behavior.

To get started, add the **testing** version of your Stripe secret to your `phpunit.xml` file:

```
<env name="STRIPE_SECRET" value="sk_test_<your-key>"/>
```

Now, whenever you interact with Cashier while testing, it will send actual API requests to your Stripe testing environment. For convenience, you should pre-fill your Stripe testing account with subscriptions / prices that you may use during testing.



In order to test a variety of billing scenarios, such as credit card denials and failures, you may use the vast range of [testing card numbers and tokens](#) provided by Stripe.

Laravel Cashier (Paddle)

- [Introduction](#)
- [Upgrading Cashier](#)
- [Installation](#)
 - [Paddle Sandbox](#)
- [Configuration](#)
 - [Billable Model](#)
 - [API Keys](#)
 - [Paddle JS](#)
 - [Currency Configuration](#)
 - [Overriding Default Models](#)
- [Quickstart](#)
 - [Selling Products](#)
 - [Selling Subscriptions](#)
- [Checkout Sessions](#)
 - [Overlay Checkout](#)
 - [Inline Checkout](#)
 - [Guest Checkouts](#)
- [Price Previews](#)
 - [Customer Price Previews](#)
 - [Discounts](#)
- [Customers](#)
 - [Customer Defaults](#)
 - [Retrieving Customers](#)
 - [Creating Customers](#)
- [Subscriptions](#)
 - [Creating Subscriptions](#)
 - [Checking Subscription Status](#)
 - [Subscription Single Charges](#)
 - [Updating Payment Information](#)
 - [Changing Plans](#)
 - [Subscription Quantity](#)
 - [Subscriptions With Multiple Products](#)
 - [Multiple Subscriptions](#)
 - [Pausing Subscriptions](#)
 - [Canceling Subscriptions](#)
- [Subscription Trials](#)
 - [With Payment Method Up Front](#)
 - [Without Payment Method Up Front](#)
 - [Extend or Activate a Trial](#)
- [Handling Paddle Webhooks](#)
 - [Defining Webhook Event Handlers](#)
 - [Verifying Webhook Signatures](#)
- [Single Charges](#)
 - [Charging for Products](#)
 - [Refunding Transactions](#)
 - [Crediting Transactions](#)
- [Transactions](#)

- [Past and Upcoming Payments](#)
- [Testing](#)

Introduction



This documentation is for Cashier Paddle 2.x's integration with Paddle Billing. If you're still using Paddle Classic, you should use [Cashier Paddle 1.x](#).

[Laravel Cashier Paddle](#) provides an expressive, fluent interface to [Paddle's](#) subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading. In addition to basic subscription management, Cashier can handle: swapping subscriptions, subscription "quantities", subscription pausing, cancelation grace periods, and more.

Before digging into Cashier Paddle, we recommend you also review Paddle's [concept guides](#) and [API documentation](#).

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review [the upgrade guide](#).

Installation

First, install the Cashier package for Paddle using the Composer package manager:

```
composer require laravel/cashier-paddle
```

Next, you should publish the Cashier migration files using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

Then, you should run your application's database migrations. The Cashier migrations will create a new `customers` table. In addition, new `subscriptions` and `subscription_items` tables will be created to store all of your customer's subscriptions. Lastly, a new `transactions` table will be created to store all of the Paddle transactions associated with your customers:

```
php artisan migrate
```



To ensure Cashier properly handles all Paddle events, remember to [set up Cashier's webhook handling](#).

Paddle Sandbox

During local and staging development, you should [register a Paddle Sandbox account](#). This account will give you a sandboxed environment to test and develop your applications without making actual payments. You may use Paddle's [test card numbers](#) to simulate various payment scenarios.

When using the Paddle Sandbox environment, you should set the `PADDLE_SANDBOX` environment variable to `true` within your application's `.env` file:

```
PADDLE_SANDBOX=true
```

After you have finished developing your application you may [apply for a Paddle vendor account](#). Before your application is placed into production, Paddle will need to approve your application's domain.

Configuration

Billable Model

Before using Cashier, you must add the `Billable` trait to your user model definition. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions and updating payment method information:

```
use Laravel\Paddle\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

If you have billable entities that are not users, you may also add the trait to those classes:

```
use Illuminate\Database\Eloquent\Model;
use Laravel\Paddle\Billable;

class Team extends Model
{
    use Billable;
}
```

API Keys

Next, you should configure your Paddle keys in your application's `.env` file. You can retrieve your Paddle API keys from the Paddle control panel:

```
PADDLE_CLIENT_SIDE_TOKEN=your-paddle-client-side-token
PADDLE_API_KEY=your-paddle-api-key
PADDLE_RETAIN_KEY=your-paddle-retain-key
PADDLE_WEBHOOK_SECRET="your-paddle-webhook-secret"
PADDLE_SANDBOX=true
```

The `PADDLE_SANDBOX` environment variable should be set to `true` when you are using [Paddle's Sandbox environment](#). The `PADDLE_SANDBOX` variable should be set to `false` if you are deploying your application to production and are using Paddle's live vendor environment.

The `PADDLE_RETAIN_KEY` is optional and should only be set if you're using Paddle with [Retain](#).

Paddle JS

Paddle relies on its own JavaScript library to initiate the Paddle checkout widget. You can load the JavaScript library by placing the `@paddleJS` Blade directive right before your application layout's closing `</head>` tag:

```
<head>
  ...
  @paddleJS
</head>
```

Currency Configuration

You can specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes [PHP's NumberFormatter class](#) to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```



In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Overriding Default Models

You are free to extend the models used internally by Cashier by defining your own model and extending the corresponding Cashier model:

```
use Laravel\Paddle\Subscription as CashierSubscription;

class Subscription extends CashierSubscription
{
    // ...
}
```

After defining your model, you may instruct Cashier to use your custom model via the [Laravel\Paddle\Cashier](#) class. Typically, you should inform Cashier about your custom models in the [boot](#) method of your application's [App\Providers\AppServiceProvider](#) class:

```
use App\Models\Cashier\Subscription;
use App\Models\Cashier\Transaction;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Cashier::useSubscriptionModel(Subscription::class);
    Cashier::useTransactionModel(Transaction::class);
}
```

Quickstart

Selling Products



Before utilizing Paddle Checkout, you should define Products with fixed prices in your Paddle dashboard. In addition, you should [configure Paddle's webhook handling](#).

Offering product and subscription billing via your application can be intimidating. However, thanks to Cashier and [Paddle's Checkout Overlay](#), you can easily build modern, robust payment integrations.

To charge customers for non-recurring, single-charge products, we'll utilize Cashier to charge customers with Paddle's Checkout Overlay, where they will provide their payment details and confirm their purchase. Once the payment has been made via the Checkout Overlay, the customer will be redirected to a success URL of your choosing within your application:

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $request->user()->checkout('pri_deluxe_album')
        ->returnTo(route('dashboard'));

    return view('buy', ['checkout' => $checkout]);
})->name('checkout');
```

As you can see in the example above, we will utilize Cashier's provided `checkout` method to create a checkout object to present the customer the Paddle Checkout Overlay for a given "price identifier". When using Paddle, "prices" refer to [defined prices for specific products](#).

If necessary, the `checkout` method will automatically create a customer in Paddle and connect that Paddle customer record to the corresponding user in your application's database. After completing the checkout session, the customer will be redirected to a dedicated success page where you can display an informational message to the customer.

In the `buy` view, we will include a button to display the Checkout Overlay. The `paddle-button` Blade component is included with Cashier Paddle; however, you may also [manually render an overlay checkout](#):

```
<x-paddle-button :checkout="$checkout" class="px-8 py-4">
    Buy Product
</x-paddle-button>
```

Providing Meta Data to Paddle Checkout

When selling products, it's common to keep track of completed orders and purchased products via `Cart` and `Order` models defined by your own application. When redirecting customers to Paddle's Checkout Overlay to complete a purchase, you may need to provide an existing order identifier so that you can associate the completed purchase with the corresponding order when the customer is redirected back to your application.

To accomplish this, you may provide an array of custom data to the `checkout` method. Let's imagine that a pending `Order` is created within our application when a user begins the checkout process. Remember, the `Cart` and `Order` models in this example are illustrative and not provided by Cashier. You are free to implement these concepts based on the needs of your own application:

```

use App\Models\Cart;
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/cart/{cart}/checkout', function (Request $request, Cart $cart) {
    $order = Order::create([
        'cart_id' => $cart->id,
        'price_ids' => $cart->price_ids,
        'status' => 'incomplete',
    ]);

    $checkout = $request->user()->checkout($order->price_ids)
        ->customData(['order_id' => $order->id]);

    return view('billing', ['checkout' => $checkout]);
})->name('checkout');

```

As you can see in the example above, when a user begins the checkout process, we will provide all of the cart / order's associated Paddle price identifiers to the `checkout` method. Of course, your application is responsible for associating these items with the "shopping cart" or order as a customer adds them. We also provide the order's ID to the Paddle Checkout Overlay via the `customData` method.

Of course, you will likely want to mark the order as "complete" once the customer has finished the checkout process. To accomplish this, you may listen to the webhooks dispatched by Paddle and raised via events by Cashier to store order information in your database.

To get started, listen for the `TransactionCompleted` event dispatched by Cashier. Typically, you should register the event listener in the `boot` method of one of your application's service providers:

```

use App\Listeners\CompleteOrder;
use Illuminate\Support\Facades\Event;
use Laravel\Paddle\Events\TransactionCompleted;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Event::listen(TransactionCompleted::class, CompleteOrder::class);
}

```

In this example, the `CompleteOrder` listener might look like the following:

```

namespace App\Listeners;

use App\Models\Order;
use Laravel\Cashier\Cashier;
use Laravel\Cashier\Events\TransactionCompleted;

class CompleteOrder
{
    /**
     * Handle the incoming Cashier webhook event.
     */
    public function handle(TransactionCompleted $event): void
    {
        $orderId = $event->payload['data']['custom_data']['order_id'] ?? null;

        $order = Order::findOrFail($orderId);

        $order->update(['status' => 'completed']);
    }
}

```

Please refer to Paddle's documentation for more information on the [data contained by the `transaction.completed` event](#).

Selling Subscriptions



Before utilizing Paddle Checkout, you should define Products with fixed prices in your Paddle dashboard. In addition, you should [configure Paddle's webhook handling](#).

Offering product and subscription billing via your application can be intimidating. However, thanks to Cashier and [Paddle's Checkout Overlay](#), you can easily build modern, robust payment integrations.

To learn how to sell subscriptions using Cashier and Paddle's Checkout Overlay, let's consider the simple scenario of a subscription service with a basic monthly (`price_basic_monthly`) and yearly (`price_basic_yearly`) plan. These two prices could be grouped under a "Basic" product (`pro_basic`) in our Paddle dashboard. In addition, our subscription service might offer an Expert plan as `pro_expert`.

First, let's discover how a customer can subscribe to our services. Of course, you can imagine the customer might click a "subscribe" button for the Basic plan on our application's pricing page. This button will invoke a Paddle Checkout Overlay for their chosen plan. To get started, let's initiate a checkout session via the `checkout` method:

```

use Illuminate\Http\Request;

Route::get('/subscribe', function (Request $request) {
    $checkout = $request->user()->checkout('price_basic_monthly')
        ->returnTo(route('dashboard'));

    return view('subscribe', ['checkout' => $checkout]);
})->name('subscribe');

```

In the `subscribe` view, we will include a button to display the Checkout Overlay. The `paddle-button` Blade component is included with Cashier Paddle; however, you may also [manually render an overlay checkout](#):

```
<x-paddle-button :checkout="$checkout" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

Now, when the Subscribe button is clicked, the customer will be able to enter their payment details and initiate their subscription. To know when their subscription has actually started (since some payment methods require a few seconds to process), you should also [configure Cashier's webhook handling](#).

Now that customers can start subscriptions, we need to restrict certain portions of our application so that only subscribed users can access them. Of course, we can always determine a user's current subscription status via the `subscribed` method provided by Cashier's `Billable` trait:

```
@if ($user->subscribed())
    <p>You are subscribed.</p>
@endif
```

We can even easily determine if a user is subscribed to specific product or price:

```
@if ($user->subscribedToProduct('pro_basic'))
    <p>You are subscribed to our Basic product.</p>
@endif

@if ($user->subscribedToPrice('price_basic_monthly'))
    <p>You are subscribed to our monthly Basic plan.</p>
@endif
```

Building a Subscribed Middleware

For convenience, you may wish to create a [middleware](#) which determines if the incoming request is from a subscribed user. Once this middleware has been defined, you may easily assign it to a route to prevent users that are not subscribed from accessing the route:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class Subscribed
{
    /**
     * Handle an incoming request.
     */
    public function handle(Request $request, Closure $next): Response
    {
        if (! $request->user()?>subscribed()) {
            // Redirect user to billing page and ask them to subscribe...
            return redirect('/subscribe');
        }

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

Once the middleware has been defined, you may assign it to a route:

```
use App\Http\Middleware\Subscribed;

Route::get('/dashboard', function () {
    // ...
})->middleware([Subscribed::class]);
```

Allowing Customers to Manage Their Billing Plan

Of course, customers may want to change their subscription plan to another product or "tier". In our example from above, we'd want to allow the customer to change their plan from a monthly subscription to a yearly subscription. For this you'll need to implement something like a button that leads to the below route:

```
use Illuminate\Http\Request;

Route::put('/subscription/{price}/swap', function (Request $request, $price) {
    $user->subscription()->swap($price); // With "$price" being "price_basic_yearly" for this example.

    return redirect()->route('dashboard');
})->name('subscription.swap');
```

Besides swapping plans you'll also need to allow your customers to cancel their subscription. Like swapping plans, provide a button that leads to the following route:

```
use Illuminate\Http\Request;

Route::put('/subscription/cancel', function (Request $request, $price) {
    $user->subscription()->cancel();

    return redirect()->route('dashboard');
})->name('subscription.cancel');
```

And now your subscription will get cancelled at the end of its billing period.



As long as you have configured Cashier's webhook handling, Cashier will automatically keep your application's Cashier-related database tables in sync by inspecting the incoming webhooks from Paddle. So, for example, when you cancel a customer's subscription via Paddle's dashboard, Cashier will receive the corresponding webhook and mark the subscription as "cancelled" in your application's database.

Checkout Sessions

Most operations to bill customers are performed using "checkouts" via Paddle's [Checkout Overlay widget](#) or by utilizing [inline checkout](#).

Before processing checkout payments using Paddle, you should define your application's [default payment link](#) in your Paddle checkout settings dashboard.

Overlay Checkout

Before displaying the Checkout Overlay widget, you must generate a checkout session using Cashier. A checkout session will inform the checkout widget of the billing operation that should be performed:

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $user->checkout('pri_34567')
        ->returnTo(route('dashboard'));

    return view('billing', ['checkout' => $checkout]);
});
```

Cashier includes a `paddle-button` [Blade component](#). You may pass the checkout session to this component as a "prop". Then, when this button is clicked, Paddle's checkout widget will be displayed:

```
<x-paddle-button :checkout="$checkout" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

By default, this will display the widget using Paddle's default styling. You can customize the widget by adding [Paddle supported attributes](#) like the `data-theme='light'` attribute to the component:

```
<x-paddle-button :url="$payLink" class="px-8 py-4" data-theme="light">
    Subscribe
</x-paddle-button>
```

The Paddle checkout widget is asynchronous. Once the user creates a subscription within the widget, Paddle will send your application a webhook so that you may properly update the subscription state in your application's database. Therefore, it's important that you properly [set up webhooks](#) to accommodate for state changes from Paddle.



After a subscription state change, the delay for receiving the corresponding webhook is typically minimal but you should account for this in your application by considering that your user's subscription might not be immediately available after completing the checkout.

Manually Rendering an Overlay Checkout

You may also manually render an overlay checkout without using Laravel's built-in Blade components. To get started, generate the checkout session [as demonstrated in previous examples](#):

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $user->checkout('pri_34567')
        ->returnTo(route('dashboard'));

    return view('billing', ['checkout' => $checkout]);
});
```

Next, you may use Paddle.js to initialize the checkout. In this example, we will create a link that is assigned the `paddle_button` class. Paddle.js will detect this class and display the overlay checkout when the link is clicked:

```
<?php
$item = $checkout->getItems();
$customer = $checkout->getCustomer();
$custom = $checkout->getCustomData();
?>

<a
    href='#!'
    class='paddle_button'
    data-items='{!! json_encode($item) !!}'
    @if ($customer) data-customer-id='{{ $customer->paddle_id }}' @endif
    @if ($custom) data-custom-data='{{ json_encode($custom) }}' @endif
    @if ($returnUrl = $checkout->getReturnUrl()) data-success-url='{{ $returnUrl }}' @endif
>
    Buy Product
</a>
```

Inline Checkout

If you don't want to make use of Paddle's "overlay" style checkout widget, Paddle also provides the option to display the widget inline. While this approach does not allow you to adjust any of the checkout's HTML fields, it allows you to embed the widget within your application.

To make it easy for you to get started with inline checkout, Cashier includes a `paddle-checkout` Blade component. To get started, you should [generate a checkout session](#):

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $user->checkout('pri_34567')
        ->returnTo(route('dashboard'));

    return view('billing', ['checkout' => $checkout]);
});
```

Then, you may pass the checkout session to the component's `checkout` attribute:

```
<x-paddle-checkout :checkout="$checkout" class="w-full" />
```

To adjust the height of the inline checkout component, you may pass the `height` attribute to the Blade component:

```
<x-paddle-checkout :checkout="$checkout" class="w-full" height="500" />
```

Please consult Paddle's [guide on Inline Checkout](#) and [available checkout settings](#) for further details on the inline checkout's customization options.

Manually Rendering an Inline Checkout

You may also manually render an inline checkout without using Laravel's built-in Blade components. To get started, generate the checkout session [as demonstrated in previous examples](#):

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $user->checkout('pri_34567')
        ->returnTo(route('dashboard'));

    return view('billing', ['checkout' => $checkout]);
});
```

Next, you may use Paddle.js to initialize the checkout. In this example, we will demonstrate this using [Alpine.js](#); however, you are free to modify this example for your own frontend stack:

```
<?php
$options = $checkout->options();

$options['settings']['frameTarget'] = 'paddle-checkout';
$options['settings']['frameInitialHeight'] = 366;
?>

<div class="paddle-checkout" x-data="{}" x-init="
    Paddle.Checkout.open(@json($options));
">
</div>
```

Guest Checkouts

Sometimes, you may need to create a checkout session for users that do not need an account with your application. To do so, you may use the `guest` method:

```
use Illuminate\Http\Request;
use Laravel\Paddle\Checkout;

Route::get('/buy', function (Request $request) {
    $checkout = Checkout::guest('pri_34567')
        ->returnTo(route('home'));

    return view('billing', ['checkout' => $checkout]);
});
```

Then, you may provide the checkout session to the [Paddle button](#) or [inline checkout](#) Blade components.

Price Previews

Paddle allows you to customize prices per currency, essentially allowing you to configure different prices for different countries. Cashier Paddle allows you to retrieve all of these prices using the `previewPrices` method. This method accepts the price IDs you wish to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::previewPrices(['pri_123', 'pri_456']);
```

The currency will be determined based on the IP address of the request; however, you may optionally provide a specific country to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices(['pri_123', 'pri_456'], ['address' => [
    'country_code' => 'BE',
    'postal_code' => '1234',
]]);
```

After retrieving the prices you may display them however you wish:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product['name'] }} - {{ $price->total() }}</li>
    @endforeach
</ul>
```

You may also display the subtotal price and tax amount separately:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->subtotal() }} (+ {{ $price->tax() }} tax)
    @endforeach
</ul>
```

For more information, [checkout Paddle's API documentation regarding price previews](#).

Customer Price Previews

If a user is already a customer and you would like to display the prices that apply to that customer, you may do so by retrieving the prices directly from the customer instance:

```
use App\Models\User;

$prices = User::find(1)->previewPrices(['pri_123', 'pri_456']);
```

Internally, Cashier will use the user's customer ID to retrieve the prices in their currency. So, for example, a user living in the United States will see prices in US dollars while a user in Belgium will see prices in Euros. If no matching currency can be found, the default currency of the product will be used. You can customize all prices of a product or subscription plan in the Paddle control panel.

Discounts

You may also choose to display prices after a discount. When calling the `previewPrices` method, you provide the discount ID via the `discount_id` option:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::previewPrices(['pri_123', 'pri_456'], [
    'discount_id' => 'dsc_123'
]);
```

Then, display the calculated prices:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product['name'] }} - {{ $price->total() }}</li>
    @endforeach
</ul>
```

Customers

Customer Defaults

Cashier allows you to define some useful defaults for your customers when creating checkout sessions. Setting these defaults allow you to pre-fill a customer's email address and name so that they can immediately move on to the payment portion of the checkout widget. You can set these defaults by overriding the following methods on your billable model:

```

/**
 * Get the customer's name to associate with Paddle.
 */
public function paddleName(): string|null
{
    return $this->name;
}

/**
 * Get the customer's email address to associate with Paddle.
 */
public function paddleEmail(): string|null
{
    return $this->email;
}

```

These defaults will be used for every action in Cashier that generates a [checkout session](#).

Retrieving Customers

You can retrieve a customer by their Paddle Customer ID using the `Cashier::findBillable` method. This method will return an instance of the billable model:

```

use Laravel\Cashier\Cashier;

$user = Cashier::findBillable($customerId);

```

Creating Customers

Occasionally, you may wish to create a Paddle customer without beginning a subscription. You may accomplish this using the `createAsCustomer` method:

```
$customer = $user->createAsCustomer();
```

An instance of `Laravel\Paddle\Customer` is returned. Once the customer has been created in Paddle, you may begin a subscription at a later date. You may provide an optional `$options` array to pass in any additional [customer creation parameters that are supported by the Paddle API](#):

```
$customer = $user->createAsCustomer($options);
```

Subscriptions

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model from your database, which will typically be an instance of `App\Models\User`. Once you have retrieved the model instance, you may use the `subscribe` method to create the model's checkout session:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $checkout = $request->user()->subscribe($premium = 12345, 'default')
        ->returnTo(route('home'));

    return view('billing', ['checkout' => $checkout]);
});
```

The first argument given to the `subscribe` method is the specific price the user is subscribing to. This value should correspond to the price's identifier in Paddle. The `returnTo` method accepts a URL that your user will be redirected to after they successfully complete the checkout. The second argument passed to the `subscribe` method should be the internal "type" of the subscription. If your application only offers a single subscription, you might call this `default` or `primary`. This subscription type is only for internal application usage and is not meant to be displayed to users. In addition, it should not contain spaces and it should never be changed after creating the subscription.

You may also provide an array of custom meta data regarding the subscription using the `customData` method:

```
$checkout = $request->user()->subscribe($premium = 12345, 'default')
->customData(['key' => 'value'])
->returnTo(route('home'));
```

Once a subscription checkout session has been created, the checkout session may be provided to the `paddle-button` [Blade component](#) that is included with Cashier Paddle:

```
<x-paddle-button :checkout="$checkout" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

After the user has finished their checkout, a `subscription_created` webhook will be dispatched from Paddle. Cashier will receive this webhook and setup the subscription for your customer. In order to make sure all webhooks are properly received and handled by your application, ensure you have properly [setup webhook handling](#).

Checking Subscription Status

Once a user is subscribed to your application, you may check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the user has an valid subscription, even if the subscription is currently within its trial period:

```
if ($user->subscribed()) {
    // ...
}
```

If your application offers multiple subscriptions, you may specify the subscription when invoking the `subscribed` method:

```
if ($user->subscribed('default')) {
    // ...
}
```

The `subscribed` method also makes a great candidate for a [route middleware](#), allowing you to filter access to routes and controllers based on the user's subscription status:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureUserIsSubscribed
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->user() && ! $request->user()->subscribed()) {
            // This user is not a paying customer...
            return redirect('billing');
        }

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

If you would like to determine if a user is still within their trial period, you may use the `onTrial` method. This method can be useful for determining if you should display a warning to the user that they are still on their trial period:

```
if ($user->subscription()->onTrial()) {
    // ...
}
```

The `subscribedToPrice` method may be used to determine if the user is subscribed to a given plan based on a given Paddle price ID. In this example, we will determine if the user's `default` subscription is actively subscribed to the monthly price:

```
if ($user->subscribedToPrice($monthly = 'pri_123', 'default')) {
    // ...
}
```

The `recurring` method may be used to determine if the user is currently on an active subscription and is no longer within their trial period or on a grace period:

```
if ($user->subscription()->recurring()) {
    // ...
}
```

Canceled Subscription Status

To determine if the user was once an active subscriber but has canceled their subscription, you may use the `canceled` method:

```
if ($user->subscription()->canceled()) {
    // ...
}
```

You may also determine if a user has canceled their subscription, but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. In addition, the `subscribed` method will still return `true` during this time:

```
if ($user->subscription()->onGracePeriod()) {
    // ...
}
```

Past Due Status

If a payment fails for a subscription, it will be marked as `past_due`. When your subscription is in this state it will not be active until the customer has updated their payment information. You may determine if a subscription is past due using the `pastDue` method on the subscription instance:

```
if ($user->subscription()->pastDue()) {
    // ...
}
```

When a subscription is past due, you should instruct the user to [update their payment information](#).

If you would like subscriptions to still be considered valid when they are `past_due`, you may use the `keepPastDueSubscriptionsActive` method provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Paddle\Cashier;

/**
 * Register any application services.
 */
public function register(): void
{
    Cashier::keepPastDueSubscriptionsActive();
}
```



When a subscription is in a `past_due` state it cannot be changed until payment information has been updated. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in a `past_due` state.

Subscription Scopes

Most subscription states are also available as query scopes so that you may easily query your database for subscriptions that are in a given state:

```
// Get all valid subscriptions...
$subscriptions = Subscription::query()->valid()->get();

// Get all of the canceled subscriptions for a user...
$subscriptions = $user->subscriptions()->canceled()->get();
```

A complete list of available scopes is available below:

```
Subscription::query()->valid();
Subscription::query()->onTrial();
Subscription::query()->expiredTrial();
Subscription::query()->notOnTrial();
Subscription::query()->active();
Subscription::query()->recurring();
Subscription::query()->pastDue();
Subscription::query()->paused();
Subscription::query()->notPaused();
Subscription::query()->onPausedGracePeriod();
Subscription::query()->notOnPausedGracePeriod();
Subscription::query()->canceled();
Subscription::query()->notCanceled();
Subscription::query()->onGracePeriod();
Subscription::query()->notOnGracePeriod();
```

Subscription Single Charges

Subscription single charges allow you to charge subscribers with a one-time charge on top of their subscriptions. You must provide one or multiple price ID's when invoking the `charge` method:

```
// Charge a single price...
$response = $user->subscription()->charge('pri_123');

// Charge multiple prices at once...
$response = $user->subscription()->charge(['pri_123', 'pri_456']);
```

The `charge` method will not actually charge the customer until the next billing interval of their subscription. If you would like to bill the customer immediately, you may use the `chargeAndInvoice` method instead:

```
$response = $user->subscription()->chargeAndInvoice('pri_123');
```

Updating Payment Information

Paddle always saves a payment method per subscription. If you want to update the default payment method for a subscription, you should redirect your customer to Paddle's hosted payment method update page using the `redirectToUpdatePaymentMethod` method on the subscription model:

```
use Illuminate\Http\Request;

Route::get('/update-payment-method', function (Request $request) {
    $user = $request->user();

    return $user->subscription()->redirectToUpdatePaymentMethod();
});
```

When a user has finished updating their information, a `subscription_updated` webhook will be dispatched by Paddle and the subscription details will be updated in your application's database.

Changing Plans

After a user has subscribed to your application, they may occasionally want to change to a new subscription plan. To update the subscription plan for a user, you should pass the Paddle price's identifier to the subscription's `swap` method:

```
use App\Models\User;

$user = User::find(1);

$user->subscription()->swap($premium = 'pri_456');
```

If you would like to swap plans and immediately invoice the user instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = User::find(1);

$user->subscription()->swapAndInvoice($premium = 'pri_456');
```

Prorations

By default, Paddle prorates charges when swapping between plans. The `noProrate` method may be used to update the subscriptions without prorating the charges:

```
$user->subscription('default')->noProrate()->swap($premium = 'pri_456');
```

If you would like to disable proration and invoice customers immediately, you may use the `swapAndInvoice` method in combination with `noProrate`:

```
$user->subscription('default')->noProrate()->swapAndInvoice($premium = 'pri_456');
```

Or, to not bill your customer for a subscription change, you may utilize the `doNotBill` method:

```
$user->subscription('default')->doNotBill()->swap($premium = 'pri_456');
```

For more information on Paddle's proration policies, please consult Paddle's [proration documentation](#).

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, a project management application might charge \$10 per month per project. To easily increment or decrement your subscription's quantity, use the `incrementQuantity` and `decrementQuantity` methods:

```
$user = User::find(1);

$user->subscription()->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription()->incrementQuantity(5);

$user->subscription()->decrementQuantity();

// Subtract five from the subscription's current quantity...
$user->subscription()->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription()->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without prorating the charges:

```
$user->subscription()->noProrate()->updateQuantity(10);
```

Quantities for Subscriptions With Multiple Products

If your subscription is a [subscription with multiple products](#), you should pass the ID of the price whose quantity you wish to increment or decrement as the second argument to the increment / decrement methods:

```
$user->subscription()->incrementQuantity(1, 'price_chat');
```

Subscriptions With Multiple Products

[Subscription with multiple products](#) allow you to assign multiple billing products to a single subscription. For example, imagine you are building a customer service "helpdesk" application that has a base subscription price of \$10 per month but offers a live chat add-on product for an additional \$15 per month.

When creating subscription checkout sessions, you may specify multiple products for a given subscription by passing an array of prices as the first argument to the `subscribe` method:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $checkout = $request->user()->subscribe([
        'price_monthly',
        'price_chat',
    ]);

    return view('billing', ['checkout' => $checkout]);
});
```

In the example above, the customer will have two prices attached to their `default` subscription. Both prices will be charged on their respective billing intervals. If necessary, you may pass an associative array of key / value pairs to indicate a specific quantity for each price:

```
$user = User::find(1);

$checkout = $user->subscribe('default', ['price_monthly', 'price_chat' => 5]);
```

If you would like to add another price to an existing subscription, you must use the subscription's `swap` method. When invoking the `swap` method, you should also include the subscription's current prices and quantities as well:

```
$user = User::find(1);

$user->subscription()->swap(['price_chat', 'price_original' => 2]);
```

The example above will add the new price, but the customer will not be billed for it until their next billing cycle. If you would like to bill the customer immediately you may use the `swapAndInvoice` method:

```
$user->subscription()->swapAndInvoice(['price_chat', 'price_original' => 2]);
```

You may remove prices from subscriptions using the `swap` method and omitting the price you want to remove:

```
$user->subscription()->swap(['price_original' => 2]);
```



You may not remove the last price on a subscription. Instead, you should simply cancel the subscription.

Multiple Subscriptions

Paddle allows your customers to have multiple subscriptions simultaneously. For example, you may run a gym that offers a swimming subscription and a weight-lifting subscription, and each subscription may have different pricing. Of course, customers should be able to subscribe to either or both plans.

When your application creates subscriptions, you may provide the type of the subscription to the `subscribe` method as the second argument. The type may be any string that represents the type of subscription the user is initiating:

```
use Illuminate\Http\Request;

Route::post('/swimming/subscribe', function (Request $request) {
    $checkout = $request->user()->subscribe($swimmingMonthly = 'pri_123', 'swimming');

    return view('billing', ['checkout' => $checkout]);
});
```

In this example, we initiated a monthly swimming subscription for the customer. However, they may want to swap to a yearly subscription at a later time. When adjusting the customer's subscription, we can simply swap the price on the `swimming` subscription:

```
$user->subscription('swimming')->swap($swimmingYearly = 'pri_456');
```

Of course, you may also cancel the subscription entirely:

```
$user->subscription('swimming')->cancel();
```

Pausing Subscriptions

To pause a subscription, call the `pause` method on the user's subscription:

```
$user->subscription()->pause();
```

When a subscription is paused, Cashier will automatically set the `paused_at` column in your database. This column is used to determine when the `paused` method should begin returning `true`. For example, if a customer pauses a subscription on March 1st, but the subscription was not scheduled to recur until March 5th, the `paused` method will continue to return `false` until March 5th. This is because a user is typically allowed to continue using an application until the end of their billing cycle.

By default, pausing happens at the next billing interval so the customer can use the remainder of the period they paid for. If you want to pause a subscription immediately, you may use the `pauseNow` method:

```
$user->subscription()->pauseNow();
```

Using the `pauseUntil` method, you can pause the subscription until a specific moment in time:

```
$user->subscription()->pauseUntil(now()->addMonth());
```

Or, you may use the `pauseNowUntil` method to immediately pause the subscription until a given point in time:

```
$user->subscription()->pauseNowUntil(now()->addMonth());
```

You may determine if a user has paused their subscription but are still on their "grace period" using the `onPausedGracePeriod` method:

```
if ($user->subscription()->onPausedGracePeriod()) {  
    // ...  
}
```

To resume a paused subscription, you may invoke the `resume` method on the subscription:

```
$user->subscription()->resume();
```



A subscription cannot be modified while it is paused. If you want to swap to a different plan or update quantities you must resume the subscription first.

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription()->cancel();
```

When a subscription is canceled, Cashier will automatically set the `ends_at` column in your database. This column is used to determine when the `subscribed` method should begin returning `false`. For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has canceled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription()->onGracePeriod()) {
    // ...
}
```

If you wish to cancel a subscription immediately, you may call the `cancelNow` method on the subscription:

```
$user->subscription()->cancelNow();
```

To stop a subscription on its grace period from canceling, you may invoke the `stopCancellation` method:

```
$user->subscription()->stopCancellation();
```



Paddle's subscriptions cannot be resumed after cancelation. If your customer wishes to resume their subscription, they will have to create a new subscription.

Subscription Trials

With Payment Method Up Front

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should set a trial time in the Paddle dashboard on the price your customer is subscribing to. Then, initiate the checkout session as normal:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $checkout = $request->user()->subscribe('pri_monthly')
        ->returnTo(route('home'));

    return view('billing', ['checkout' => $checkout]);
});
```

When your application receives the `subscription_created` event, Cashier will set the trial period ending date on the subscription record within your application's database as well as instruct Paddle to not begin billing the customer until after this date.



If the customer's subscription is not canceled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

You may determine if the user is within their trial period using either the `onTrial` method of the user instance or the `onTrial` method of the subscription instance. The two examples below are equivalent:

```
if ($user->onTrial()) {
    // ...
}

if ($user->subscription()->onTrial()) {
    // ...
}
```

To determine if an existing trial has expired, you may use the `hasExpiredTrial` methods:

```
if ($user->hasExpiredTrial()) {
    // ...
}

if ($user->subscription()->hasExpiredTrial()) {
    // ...
}
```

To determine if a user is on trial for a specific subscription type, you may provide the type to the `onTrial` or `hasExpiredTrial` methods:

```
if ($user->onTrial('default')) {
    // ...
}

if ($user->hasExpiredTrial('default')) {
    // ...
}
```

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the customer record attached to your user to your desired trial ending date. This is typically done during user registration:

```
use App\Models\User;

$user = User::create([
    // ...
]);

$user->createAsCustomer([
    'trial_ends_at' => now()->addDays(10)
]);
```

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the `User` instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `subscribe` method as usual:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $checkout = $user->subscribe('pri_monthly')
        ->returnTo(route('home'));

    return view('billing', ['checkout' => $checkout]);
});
```

To retrieve the user's trial ending date, you may use the `trialEndsAt` method. This method will return a Carbon date instance if a user is on a trial or `null` if they aren't. You may also pass an optional subscription type parameter if you would like to get the trial ending date for a specific subscription other than the default one:

```
if ($user->onTrial('default')) {
    $trialEndsAt = $user->trialEndsAt();
}
```

You may use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not created an actual subscription yet:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

Extend or Activate a Trial

You can extend an existing trial period on a subscription by invoking the `extendTrial` method and specifying the moment in time that the trial should end:

```
$user->subscription()->extendTrial(now()->addDays(5));
```

Or, you may immediately activate a subscription by ending its trial by calling the `activate` method on the subscription:

```
$user->subscription()->activate();
```

Handling Paddle Webhooks

Paddle can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is registered by the Cashier service provider. This controller will handle all incoming webhook requests.

By default, this controller will automatically handle canceling subscriptions that have too many failed charges, subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any Paddle webhook event you like.

To ensure your application can handle Paddle webhooks, be sure to [configure the webhook URL in the Paddle control panel](#). By default, Cashier's webhook controller responds to the `/paddle/webhook` URL path. The full list of all webhooks

you should enable in the Paddle control panel are:

- Customer Updated
- Transaction Completed
- Transaction Updated
- Subscription Created
- Subscription Updated
- Subscription Paused
- Subscription Canceled



Make sure you protect incoming requests with Cashier's included [webhook signature verification middleware](#).

Webhooks and CSRF Protection

Since Paddle webhooks need to bypass Laravel's [CSRF protection](#), be sure to list the URI as an exception in your `App\Http\Middleware\VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'paddle/*',
];
```

Webhooks and Local Development

For Paddle to be able to send your application webhooks during local development, you will need to expose your application via a site sharing service such as [Ngrok](#) or [Expose](#). If you are developing your application locally using [Laravel Sail](#), you may use Sail's [site sharing command](#).

Defining Webhook Event Handlers

Cashier automatically handles subscription cancelation on failed charges and other common Paddle webhooks. However, if you have additional webhook events you would like to handle, you may do so by listening to the following events that are dispatched by Cashier:

- `Laravel\Paddle\Events\WebhookReceived`
- `Laravel\Paddle\Events\WebhookHandled`

Both events contain the full payload of the Paddle webhook. For example, if you wish to handle the `transaction_billed` webhook, you may register a [listener](#) that will handle the event:

```
<?php

namespace App\Listeners;

use Laravel\Paddle\Events\WebhookReceived;

class PaddleEventListener
{
    /**
     * Handle received Paddle webhooks.
     */
    public function handle(WebhookReceived $event): void
    {
        if ($event->payload['alert_name'] === 'transaction_billed') {
            // Handle the incoming event...
        }
    }
}
```

Once your listener has been defined, you may register it within your application's `EventServiceProvider`:

```
<?php

namespace App\Providers;

use App\Listeners\PaddleEventListener;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Laravel\Paddle\Events\WebhookReceived;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        WebhookReceived::class => [
            PaddleEventListener::class,
        ],
    ];
}
```

Cashier also emit events dedicated to the type of the received webhook. In addition to the full payload from Paddle, they also contain the relevant models that were used to process the webhook such as the billable model, the subscription, or the receipt:

- `Laravel\Paddle\Events\CustomerUpdated`
- `Laravel\Paddle\Events\TransactionCompleted`
- `Laravel\Paddle\Events\TransactionUpdated`
- `Laravel\Paddle\Events\SubscriptionCreated`
- `Laravel\Paddle\Events\SubscriptionUpdated`
- `Laravel\Paddle\Events\SubscriptionPaused`
- `Laravel\Paddle\Events\SubscriptionCanceled`

You can also override the default, built-in webhook route by defining the `CASHIER_WEBHOOK` environment variable in your application's `.env` file. This value should be the full URL to your webhook route and needs to match the URL set in your Paddle control panel:

```
CASHIER_WEBHOOK=https://example.com/my-paddle-webhook-url
```

Verifying Webhook Signatures

To secure your webhooks, you may use [Paddle's webhook signatures](#). For convenience, Cashier automatically includes a middleware which validates that the incoming Paddle webhook request is valid.

To enable webhook verification, ensure that the `PADDLE_WEBHOOK_SECRET` environment variable is defined in your application's `.env` file. The webhook secret may be retrieved from your Paddle account dashboard.

Single Charges

Charging for Products

If you would like to initiate a product purchase for a customer, you may use the `checkout` method on a billable model instance to generate a checkout session for the purchase. The `checkout` method accepts one or multiple price ID's. If necessary, an associative array may be used to provide the quantity of the product that is being purchased:

```
use Illuminate\Http\Request;

Route::get('/buy', function (Request $request) {
    $checkout = $request->user()->checkout(['pri_tshirt', 'pri_socks' => 5]);

    return view('buy', ['checkout' => $checkout]);
});
```

After generating the checkout session, you may use Cashier's provided `paddle-button` Blade component to allow the user to view the Paddle checkout widget and complete the purchase:

```
<x-paddle-button :checkout="$checkout" class="px-8 py-4">
    Buy
</x-paddle-button>
```

A checkout session has a `customData` method, allowing you to pass any custom data you wish to the underlying transaction creation. Please consult [the Paddle documentation](#) to learn more about the options available to you when passing custom data:

```
$checkout = $user->checkout('pri_tshirt')
->customData([
    'custom_option' => $value,
]);
```

Refunding Transactions

Refunding transactions will return the refunded amount to your customer's payment method that was used at the time of purchase. If you need to refund a Paddle purchase, you may use the `refund` method on a `Cashier\Paddle\Transaction` model. This method accepts a reason as the first argument, one or more price ID's to refund with optional amounts as an associative array. You may retrieve the transactions for a given billable model using the `transactions` method.

For example, imagine we want to refund a specific transaction for prices `pri_123` and `pri_456`. We want to fully refund `pri_123`, but only refund two dollars for `pri_456`:

```
use App\Models\User;

$user = User::find(1);

$transaction = $user->transactions()->first();

$response = $transaction->refund('Accidental charge', [
    'pri_123', // Fully refund this price...
    'pri_456' => 200, // Only partially refund this price...
]);
```

The example above refunds specific line items in a transaction. If you want to refund the entire transaction, simply provide a reason:

```
$response = $transaction->refund('Accidental charge');
```

For more information on refunds, please consult [Paddle's refund documentation](#).



Refunds must always be approved by Paddle before fully processing.

Crediting Transactions

Just like refunding, you can also credit transactions. Crediting transactions will add the funds to the customer's balance so it may be used for future purchases. Crediting transactions can only be done for manually-collected transactions and not for automatically-collected transactions (like subscriptions) since Paddle handles subscription credits automatically:

```
$transaction = $user->transactions()->first();

// Credit a specific line item fully...
$response = $transaction->credit('Compensation', 'pri_123');
```

For more info, [see Paddle's documentation on crediting](#).



Credits can only be applied for manually-collected transactions. Automatically-collected transactions are credited by Paddle themselves.

Transactions

You may easily retrieve an array of a billable model's transactions via the `transactions` property:

```
use App\Models\User;

$user = User::find(1);

$transactions = $user->transactions;
```

Transactions represent payments for your products and purchases and are accompanied by invoices. Only completed transactions are stored in your application's database.

When listing the transactions for a customer, you may use the transaction instance's methods to display the relevant payment information. For example, you may wish to list every transaction in a table, allowing the user to easily download any of the invoices:

```
<table>
    @foreach ($transactions as $transaction)
        <tr>
            <td>{{ $transaction->billed_at->toFormattedDateString() }}</td>
            <td>{{ $transaction->total() }}</td>
            <td>{{ $transaction->tax() }}</td>
            <td><a href="{{ route('download-invoice', $transaction->id) }}" target="_blank">Download</a></td>
        </tr>
    @endforeach
</table>
```

The `download-invoice` route may look like the following:

```
use Illuminate\Http\Request;
use Laravel\Cashier\Transaction;

Route::get('/download-invoice/{transaction}', function (Request $request, Transaction $transaction) {
    return $transaction->redirectToInvoicePdf();
})->name('download-invoice');
```

Past and Upcoming Payments

You may use the `lastPayment` and `nextPayment` methods to retrieve and display a customer's past or upcoming payments for recurring subscriptions:

```
use App\Models\User;

$user = User::find(1);

$subscription = $user->subscription();

$lastPayment = $subscription->lastPayment();
$nextPayment = $subscription->nextPayment();
```

Both of these methods will return an instance of `Laravel\Paddle\Payment`; however, `lastPayment` will return `null` when transactions have not been synced by webhooks yet, while `nextPayment` will return `null` when the billing cycle has ended (such as when a subscription has been canceled):

```
Next payment: {{ $nextPayment->amount() }} due on {{ $nextPayment->date()->format('d/m/Y') }}
```

Testing

While testing, you should manually test your billing flow to make sure your integration works as expected.

For automated tests, including those executed within a CI environment, you may use [Laravel's HTTP Client](#) to fake HTTP calls made to Paddle. Although this does not test the actual responses from Paddle, it does provide a way to test your application without actually calling Paddle's API.

Laravel Dusk

- [Introduction](#)
- [Installation](#)
 - [Managing ChromeDriver Installations](#)
 - [Using Other Browsers](#)
- [Getting Started](#)
 - [Generating Tests](#)
 - [Resetting the Database After Each Test](#)
 - [Running Tests](#)
 - [Environment Handling](#)
- [Browser Basics](#)
 - [Creating Browsers](#)
 - [Navigation](#)
 - [Resizing Browser Windows](#)
 - [Browser Macros](#)
 - [Authentication](#)
 - [Cookies](#)
 - [Executing JavaScript](#)
 - [Taking a Screenshot](#)
 - [Storing Console Output to Disk](#)
 - [Storing Page Source to Disk](#)
- [Interacting With Elements](#)
 - [Dusk Selectors](#)
 - [Text, Values, and Attributes](#)
 - [Interacting With Forms](#)
 - [Attaching Files](#)
 - [Pressing Buttons](#)
 - [Clicking Links](#)
 - [Using the Keyboard](#)
 - [Using the Mouse](#)
 - [JavaScript Dialogs](#)
 - [Interacting With Inline Frames](#)
 - [Scoping Selectors](#)
 - [Waiting for Elements](#)
 - [Scrolling an Element Into View](#)
- [Available Assertions](#)
- [Pages](#)
 - [Generating Pages](#)
 - [Configuring Pages](#)
 - [Navigating to Pages](#)
 - [Shorthand Selectors](#)
 - [Page Methods](#)
- [Components](#)
 - [Generating Components](#)
 - [Using Components](#)
- [Continuous Integration](#)
 - [Heroku CI](#)
 - [Travis CI](#)
 - [GitHub Actions](#)

- o [Chipper CI](#)

Introduction

[Laravel Dusk](#) provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your local computer. Instead, Dusk uses a standalone [ChromeDriver](#) installation. However, you are free to utilize any other Selenium compatible driver you wish.

Installation

To get started, you should install [Google Chrome](#) and add the `laravel/dusk` Composer dependency to your project:

```
composer require laravel/dusk --dev
```



*If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.*

After installing the Dusk package, execute the `dusk:install` Artisan command. The `dusk:install` command will create a `tests/Browser` directory, an example Dusk test, and install the Chrome Driver binary for your operating system:

```
php artisan dusk:install
```

Next, set the `APP_URL` environment variable in your application's `.env` file. This value should match the URL you use to access your application in a browser.



If you are using [Laravel Sail](#) to manage your local development environment, please also consult the [Sail documentation](#) on [configuring and running Dusk tests](#).

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is installed by Laravel Dusk via the `dusk:install` command, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 86

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all

# Install the version of ChromeDriver that matches the detected version of Chrome / Chromium for
# your OS...
php artisan dusk:chrome-driver --detect
```



Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone `ChromeDriver` installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the `ChromeDriver`:

```
/** 
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 */
public static function prepare(): void
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;

/**
 * Create the RemoteWebDriver instance.
 */
protected function driver(): RemoteWebDriver
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Resetting the Database After Each Test

Most of the tests you write will interact with pages that retrieve data from your application's database; however, your Dusk tests should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which

will not be applicable or available across HTTP requests. Instead, you have two options: the `DatabaseMigrations` trait and the `DatabaseTruncation` trait.

Using Database Migrations

The `DatabaseMigrations` trait will run your database migrations before each test. However, dropping and re-creating your database tables for each test is typically slower than truncating the tables:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

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



SQLite in-memory databases may not be used when executing Dusk tests. Since the browser executes within its own process, it will not be able to access the in-memory databases of other processes.

Using Database Truncation

Before using the `DatabaseTruncation` trait, you must install the `doctrine/dbal` package using the Composer package manager:

```
composer require --dev doctrine/dbal
```

The `DatabaseTruncation` trait will migrate your database on the first test in order to ensure your database tables have been properly created. However, on subsequent tests, the database's tables will simply be truncated - providing a speed boost over re-running all of your database migrations:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseTruncation;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseTruncation;
}
```

By default, this trait will truncate all tables except the `migrations` table. If you would like to customize the tables that should be truncated, you may define a `$tablesToTruncate` property on your test class:

```
/**
 * Indicates which tables should be truncated.
 *
 * @var array
 */
protected $tablesToTruncate = ['users'];
```

Alternatively, you may define an `$exceptTables` property on your test class to specify which tables should be excluded from truncation:

```
/**
 * Indicates which tables should be excluded from truncation.
 *
 * @var array
 */
protected $exceptTables = ['users'];
```

To specify the database connections that should have their tables truncated, you may define a `$connectionsToTruncate` property on your test class:

```
/**
 * Indicates which connections should have their tables truncated.
 *
 * @var array
 */
protected $connectionsToTruncate = ['mysql'];
```

If you would like to execute code before or after database truncation is performed, you may define `beforeTruncatingDatabase` or `afterTruncatingDatabase` methods on your test class:

```
/**
 * Perform any work that should take place before the database has started truncating.
 */
protected function beforeTruncatingDatabase(): void
{
    //
}

/**
 * Perform any work that should take place after the database has finished truncating.
 */
protected function afterTruncatingDatabase(): void
{
    //
}
```

Running Tests

To run your browser tests, execute the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, such as allowing you to only run the tests for a given [group](#):

```
php artisan dusk --group=foo
```



If you are using [Laravel Sail](#) to manage your local development environment, please consult the [Sail](#) documentation on [configuring and running Dusk tests](#).

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 */  
public static function prepare(): void  
{  
    // static::startChromeDriver();  
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class to reflect the correct port:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;  
  
/**  
 * Create the RemoteWebDriver instance.  
 */  
protected function driver(): RemoteWebDriver  
{  
    return RemoteWebDriver::create(  
        'http://localhost:9515', DesiredCapabilities::chrome()  
    );  
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a

.env.dusk.local file.

When running tests, Dusk will back-up your .env file and rename your Dusk environment to .env. Once the tests have completed, your .env file will be restored.

Browser Basics

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, you may call the `browse` method from within your Dusk test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     */
    public function test_basic_example(): void
    {
        $user = User::factory()->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function (Browser $browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'password')
                ->press('Login')
                ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a closure. A browser instance will automatically be passed to this closure by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, simply add more browser arguments to the signature of the closure given to the `browse` method:

```
$this->browse(function (Browser $first, Browser $second) {
    $first->loginAs(User::find(1))
        ->visit('/home')
        ->waitForText('Message');

    $second->loginAs(User::find(2))
        ->visit('/home')
        ->waitForText('Message')
        ->type('message', 'Hey Taylor')
        ->press('Send');

    $first->waitForText('Hey Taylor')
        ->assertSee('Jeffrey Way');
});
```

Navigation

The `visit` method may be used to navigate to a given URI within your application:

```
$browser->visit('/login');
```

You may use the `visitRoute` method to navigate to a [named route](#):

```
$browser->visitRoute('login');
```

You may navigate "back" and "forward" using the `back` and `forward` methods:

```
$browser->back();
$browser->forward();
```

You may use the `refresh` method to refresh the page:

```
$browser->refresh();
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

The `fitContent` method will resize the browser window to match the size of its content:

```
$browser->fitContent();
```

When a test fails, Dusk will automatically resize the browser to fit the content prior to taking a screenshot. You may disable this feature by calling the `disableFitOnFailure` method within your test:

```
$browser->disableFitOnFailure();
```

You may use the `move` method to move the browser window to a different position on your screen:

```
$browser->move($x = 100, $y = 100);
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a [service provider's](#) `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     */
    public function boot(): void
    {
        Browser::macro('scrollToElement', function (string $element = null) {
            $this->script("$(`html, body`).animate({ scrollTop: `#${$element}`.offset().top }, 0);");

            return $this;
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Browser` instance:

```
$this->browse(function (Browser $browser) use ($user) {
    $browser->visit('/pay')
        ->scrollToElement('#credit-card-details')
        ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with your application's login screen during every test. The `loginAs` method accepts a primary key associated with your authenticatable model or an authenticatable model instance:

```
use App\Models\User;
use Laravel\Dusk\Browser;

$this->browse(function (Browser $browser) {
    $browser->loginAs(User::find(1))
        ->visit('/home');
});
```



After using the `loginAs` method, the user session will be maintained for all tests within the file.

Cookies

You may use the `cookie` method to get or set an encrypted cookie's value. By default, all of the cookies created by Laravel are encrypted:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

You may use the `plainCookie` method to get or set an unencrypted cookie's value:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

You may use the `deleteCookie` method to delete the given cookie:

```
$browser->deleteCookie('name');
```

Executing JavaScript

You may use the `script` method to execute arbitrary JavaScript statements within the browser:

```
$browser->script('document.documentElement.scrollTop = 0');

$browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);

$output = $browser->script('return window.location.pathname');
```

Taking a Screenshot

You may use the `screenshot` method to take a screenshot and store it with the given filename. All screenshots will be stored within the `tests/Browser/screenshots` directory:

```
$browser->screenshot('filename');
```

The `responsiveScreenshots` method may be used to take a series of screenshots at various breakpoints:

```
$browser->responsiveScreenshots('filename');
```

Storing Console Output to Disk

You may use the `storeConsoleLog` method to write the current browser's console output to disk with the given filename. Console output will be stored within the `tests/Browser/console` directory:

```
$browser->storeConsoleLog('filename');
```

Storing Page Source to Disk

You may use the `storeSource` method to write the current page's source to disk with the given filename. The page source will be stored within the `tests/Browser/source` directory:

```
$browser->storeSource('filename');
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...
<button>Login</button>
// Test...
$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, when interacting with a Dusk browser, prefix the selector with `@` to manipulate the attached element within your test:

```
// HTML...
<button dusk="login-button">Login</button>
// Test...
$browser->click('@login-button');
```

If desired, you may customize the HTML attribute that the Dusk selector utilizes via the `selectorHtmlAttribute` method. Typically, this method should be called from the `boot` method of your application's `AppServiceProvider`:

```
use Laravel\Dusk\Dusk;
Dusk::selectorHtmlAttribute('data-dusk');
```

Text, Values, and Attributes

Retrieving and Setting Values

Dusk provides several methods for interacting with the current value, display text, and attributes of elements on the page. For example, to get the "value" of an element that matches a given CSS or Dusk selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

You may use the `inputValue` method to get the "value" of an input element that has a given field name:

```
$value = $browser->inputValue('field');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve the value of an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Interacting With Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an `input` or `textarea` field with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
    ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

You can instruct Dusk to type slowly using the `typeSlowly` method. By default, Dusk will pause for 100 milliseconds between key presses. To customize the amount of time between key presses, you may pass the appropriate number of

milliseconds as the third argument to the method:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');
$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

You may use the `appendSlowly` method to append text slowly:

```
$browser->type('tags', 'foo')
->appendSlowly('tags', 'bar', 'baz');
```

Dropdowns

To select a value available on a `select` element, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second argument:

```
$browser->select('size');
```

By providing an array as the second argument to the `select` method, you can instruct the method to select multiple options:

```
$browser->select('categories', ['Art', 'Music']);
```

Checkboxes

To "check" a checkbox input, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');
```

The `uncheck` method may be used to "uncheck" a checkbox input:

```
$browser->uncheck('terms');
```

Radio Buttons

To "select" a `radio` input option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `radio` input with matching `name` and `value` attributes:

```
$browser->radio('size', 'large');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `file` input with a matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```



The `attach` function requires the `Zip` PHP extension to be installed and enabled on your server.

Pressing Buttons

The `press` method may be used to click a button element on the page. The argument given to the `press` method may be either the display text of the button or a CSS / Dusk selector:

```
$browser->press('Login');
```

When submitting forms, many applications disable the form's submission button after it is pressed and then re-enable the button when the form submission's HTTP request is complete. To press a button and wait for the button to be re-enabled, you may use the `pressAndwaitFor` method:

```
// Press the button and wait a maximum of 5 seconds for it to be enabled...
$browser->pressAndwaitFor('Save');

// Press the button and wait a maximum of 1 second for it to be enabled...
$browser->pressAndwaitFor('Save', 1);
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

You may use the `seeLink` method to determine if a link with the given display text is visible on the page:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```



These methods interact with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Using the Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may instruct Dusk to hold modifier keys while entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `swift` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Another valuable use case for the `keys` method is sending a "keyboard shortcut" combination to the primary CSS selector for your application:

```
$browser->keys('.app', ['{command}', 'j']);
```



All modifier keys such as `{command}` are wrapped in `[]` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be [found on GitHub](#).

Fluent Keyboard Interactions

Dusk also provides a `withKeyboard` method, allowing you to fluently perform complex keyboard interactions via the `Laravel\Dusk\Keyboard` class. The `Keyboard` class provides `press`, `release`, `type`, and `pause` methods:

```
use Laravel\Dusk\Keyboard;

$browser->withKeyboard(function (Keyboard $keyboard) {
    $keyboard->press('c')
        ->pause(1000)
        ->release('c')
        ->type(['c', 'e', 'o']);
});
```

Keyboard Macros

If you would like to define custom keyboard interactions that you can easily re-use throughout your test suite, you may use the `macro` method provided by the `Keyboard` class. Typically, you should call this method from a [service provider's boot](#) method:

```
<?php

namespace App\Providers;

use Facebook\WebDriver\WebDriverKeys;
use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Keyboard;
use Laravel\Dusk\OperatingSystem;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     */
    public function boot(): void
    {
        Keyboard::macro('copy', function (string $element = null) {
            $this->type([
                OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTROL, 'c',
            ]);
        });

        return $this;
    }

    Keyboard::macro('paste', function (string $element = null) {
        $this->type([
            OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTROL, 'v',
        ]);
    });

    return $this;
}
}
```

The `macro` function accepts a name as its first argument and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Keyboard` instance:

```
$browser->click('@textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->copy())
->click('@another-textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->paste());
```

Using the Mouse

Clicking on Elements

The `click` method may be used to click on an element matching the given CSS or Dusk selector:

```
$browser->click('.selector');
```

The `clickAtXPath` method may be used to click on an element matching the given XPath expression:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

The `clickAtPoint` method may be used to click on the topmost element at a given pair of coordinates relative to the viewable area of the browser:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

The `doubleClick` method may be used to simulate the double click of a mouse:

```
$browser->doubleClick();  
  
$browser->doubleClick('.selector');
```

The `rightClick` method may be used to simulate the right click of a mouse:

```
$browser->rightClick();  
  
$browser->rightClick('.selector');
```

The `clickAndHold` method may be used to simulate a mouse button being clicked and held down. A subsequent call to the `releaseMouse` method will undo this behavior and release the mouse button:

```
$browser->clickAndHold('.selector');  
  
$browser->clickAndHold()  
    ->pause(1000)  
    ->releaseMouse();
```

The `controlClick` method may be used to simulate the `ctrl+click` event within the browser:

```
$browser->controlClick();  
  
$browser->controlClick('.selector');
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given CSS or Dusk selector:

```
$browser->mouseover('.selector');
```

Drag and Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', $pixels = 10);  
$browser->dragRight('.selector', $pixels = 10);  
$browser->dragUp('.selector', $pixels = 10);  
$browser->dragDown('.selector', $pixels = 10);
```

Finally, you may drag an element by a given offset:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs. For example, you may use the `waitForDialog` method to wait for a JavaScript dialog to appear. This method accepts an optional argument indicating how many seconds to wait for the dialog to appear:

```
$browser->waitForDialog($seconds = null);
```

The `assertDialogOpened` method may be used to assert that a dialog has been displayed and contains the given message:

```
$browser->assertDialogOpened('Dialog message');
```

If the JavaScript dialog contains a prompt, you may use the `typeInDialog` method to type a value into the prompt:

```
$browser->typeInDialog('Hello World');
```

To close an open JavaScript dialog by clicking the "OK" button, you may invoke the `acceptDialog` method:

```
$browser->acceptDialog();
```

To close an open JavaScript dialog by clicking the "Cancel" button, you may invoke the `dismissDialog` method:

```
$browser->dismissDialog();
```

Interacting With Inline Frames

If you need to interact with elements within an iframe, you may use the `withinFrame` method. All element interactions that take place within the closure provided to the `withinFrame` method will be scoped to the context of the specified iframe:

```
$browser->withinFrame('#credit-card-details', function ($browser) {
    $browser->type('input[name="cardnumber"]', '4242424242424242')
        ->type('input[name="exp-date"]', '12/24')
        ->type('input[name="cvc"]', '123');
    })->press('Pay');
});
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to accomplish this. All operations performed within the closure given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function (Browser $table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

You may occasionally need to execute assertions outside of the current scope. You may use the `elsewhere` and `elsewhereWhenAvailable` methods to accomplish this:

```
$browser->with('.table', function (Browser $table) {
    // Current scope is `body .table`...

    $browser->elsewhere('.page-title', function (Browser $title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });

    $browser->elsewhereWhenAvailable('.page-title', function (Browser $title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });
});
```

Waiting for Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to become visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you just need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

If you need to pause the test only if a given condition is `true`, use the `pauseIf` method:

```
$browser->pauseIf(App::environment('production'), 1000);
```

Likewise, if you need to pause the test unless a given condition is `true`, you may use the `pauseUnless` method:

```
$browser->pauseUnless(App::environment('testing'), 1000);
```

Waiting for Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS or Dusk selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

You may also wait until the element matching the given selector contains the given text:

```
// Wait a maximum of five seconds for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World');

// Wait a maximum of one second for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

You may also wait until the element matching the given selector is missing from the page:

```
// Wait a maximum of five seconds until the selector is missing...
$browser->waitUntilMissing('.selector');

// Wait a maximum of one second until the selector is missing...
$browser->waitUntilMissing('.selector', 1);
```

Or, you may wait until the element matching the given selector is enabled or disabled:

```
// Wait a maximum of five seconds until the selector is enabled...
$browser->waitUntilEnabled('.selector');

// Wait a maximum of one second until the selector is enabled...
$browser->waitUntilEnabled('.selector', 1);

// Wait a maximum of five seconds until the selector is disabled...
$browser->waitUntilDisabled('.selector');

// Wait a maximum of one second until the selector is disabled...
$browser->waitUntilDisabled('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for an element to appear that matches a given selector and then interact with the element. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used to accomplish this. All element operations performed within the given closure will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function (Browser $modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

Waiting for Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

You may use the `waitUntilMissingText` method to wait until the displayed text has been removed from the page:

```
// Wait a maximum of five seconds for the text to be removed...
$browser->waitForMissingText('Hello World');

// Wait a maximum of one second for the text to be removed...
$browser->waitForMissingText('Hello World', 1);
```

Waiting for Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting for Inputs

The `waitForInput` method may be used to wait until the given input field is visible on the page:

```
// Wait a maximum of five seconds for the input...
$browser->waitForInput($field);

// Wait a maximum of one second for the input...
$browser->waitForInput($field, 1);
```

Waiting on the Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

The `waitForLocation` method can also be used to wait for the current window location to be a fully qualified URL:

```
$browser->waitForLocation('https://example.com/path');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to wait for a page to reload after performing an action, use the `waitForReload` method:

```
use Laravel\Dusk\Browser;

$browser->waitForReload(function (Browser $browser) {
    $browser->press('Submit');
})
->assertSee('Success!');
```

Since the need to wait for the page to reload typically occurs after clicking a button, you may use the `clickAndwaitForReload` method for convenience:

```
$browser->clickAndwaitForReload('.selector')
    ->assertSee('something');
```

Waiting on JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitFor` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitFor('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitFor('App.data.servers.length > 0', 1);
```

Waiting on Vue Expressions

The `waitForVue` and `waitForVueIsNot` methods may be used to wait until a Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitForVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitForVueIsNot('user.name', null, '@user');
```

Waiting for JavaScript Events

The `waitForEvent` method can be used to pause the execution of a test until a JavaScript event occurs:

```
$browser->waitForEvent('load');
```

The event listener is attached to the current scope, which is the `body` element by default. When using a scoped selector, the event listener will be attached to the matching element:

```
$browser->with('iframe', function (Browser $iframe) {
    // Wait for the iframe's load event...
    $iframe->waitForEvent('load');
});
```

You may also provide a selector as the second argument to the `waitForEvent` method to attach the event listener to a specific element:

```
$browser->waitForEvent('load', '.selector');
```

You may also wait for events on the `document` and `window` objects:

```
// Wait until the document is scrolled...
$browser->waitForEvent('scroll', 'document');

// Wait a maximum of five seconds until the window is resized...
$browser->waitForEvent('resize', 'window', 5);
```

Waiting With a Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given closure to return `true`. The `waitUsing` method accepts the maximum number of seconds to wait, the interval at which the closure should be evaluated, the closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Scrolling an Element Into View

Sometimes you may not be able to click on an element because it is outside of the viewable area of the browser. The `scrollIntoView` method will scroll the browser window until the element at the given selector is within the view:

```
$browser->scrollIntoView('.selector')
->click('.selector');
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

assertTitle	assertDontSee	assertAttribute
assertTitleContains	assertSeeln	assertAttributeContains
assertUrls	assertDontSeeln	assertAttributeDoesntContain
assertSchemes	assertSeeAnythingIn	assertAriaAttribute
assertSchemesNot	assertSeeNothingIn	assertDataAttribute
assertHostIs	assertScript	assertVisible
assertHostIsNot	assertSourceHas	assertPresent
assertPortIs	assertSourceMissing	assertNotPresent
assertPortIsNot	assertSeeLink	assertMissing
assertPathBeginsWith	assertDontSeeLink	assertInputPresent
assertPathIs	assertInputValue	assertInputMissing
assertPathIsNot	assertInputValuesNot	assertDialogOpened
assertRoutes	assertChecked	assertEnabled
assertQueryStringHas	assertNotChecked	assertDisabled
assertQueryStringMissing	assertIndeterminate	assertButtonEnabled
assertFragments	assertRadioSelected	assertButtonDisabled
assertFragmentBeginsWith	assertRadioNotSelected	assertFocused
assertFragmentIsNot	assertSelected	assertNotFocused
assertHasCookie	assertNotSelected	assertAuthenticated
assertHasPlainCookie	assertSelectHasOptions	assertGuest
assertCookieMissing	assertSelectMissingOptions	assertAuthenticatedAs
assertPlainCookieMissing	assertSelectHasOption	assertVue
assertCookieValue	assertSelectMissingOption	assertVuelsNot
assertPlainCookieValue	assertValue	assertVueContains
assertSee	assertValuesNot	assertVueDoesntContain

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith('/home');
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the URL's current hash fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the URL's current hash fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the URL's current hash fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given encrypted cookie is present:

```
$browser->assertHasCookie($name);
```

assertHasPlainCookie

Assert that the given unencrypted cookie is present:

```
$browser->assertHasPlainCookie($name);
```

assertCookieMissing

Assert that the given encrypted cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertPlainCookieMissing

Assert that the given unencrypted cookie is not present:

```
$browser->assertPlainCookieMissing($name);
```

assertCookieValue

Assert that an encrypted cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSeeAnythingIn

Assert that any text is present within the selector:

```
$browser->assertSeeAnythingIn($selector);
```

assertSeeNothingIn

Assert that no text is present within the selector:

```
$browser->assertSeeNothingIn($selector);
```

assertScript

Assert that the given JavaScript expression evaluates to the given value:

```
$browser->assertScript('window.isLoaded')
    ->assertScript('document.readyState', 'complete');
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValuesNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertIndeterminate

Assert that the given checkbox is in an indeterminate state:

```
$browser->assertIndeterminate($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertSelectMissingOption

Assert that the given value is not available to be selected:

```
$browser->assertSelectMissingOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertValueIsNot

Assert that the element matching the given selector does not have the given value:

```
$browser->assertValueIsNot($selector, $value);
```

assertAttribute

Assert that the element matching the given selector has the given value in the provided attribute:

```
$browser->assertAttribute($selector, $attribute, $value);
```

assertAttributeContains

Assert that the element matching the given selector contains the given value in the provided attribute:

```
$browser->assertAttributeContains($selector, $attribute, $value);
```

assertAttributeDoesntContain

Assert that the element matching the given selector does not contain the given value in the provided attribute:

```
$browser->assertAttributeDoesntContain($selector, $attribute, $value);
```

assertAriaAttribute

Assert that the element matching the given selector has the given value in the provided aria attribute:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

For example, given the markup `<button aria-label="Add"></button>`, you may assert against the `aria-label` attribute like so:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

assertDataAttribute

Assert that the element matching the given selector has the given value in the provided data attribute:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

For example, given the markup `<tr id="row-1" data-content="attendees"></tr>`, you may assert against the `data-label` attribute like so:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present in the source:

```
$browser->assertPresent($selector);
```

assertNotPresent

Assert that the element matching the given selector is not present in the source:

```
$browser->assertNotPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertInputPresent

Assert that an input with the given name is present:

```
$browser->assertInputPresent($name);
```

assertInputMissing

Assert that an input with the given name is not present in the source:

```
$browser->assertInputMissing($name);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertAuthenticated

Assert that the user is authenticated:

```
$browser->assertAuthenticated();
```

assertGuest

Assert that the user is not authenticated:

```
$browser->assertGuest();
```

assertAuthenticatedAs

Assert that the user is authenticated as the given user:

```
$browser->assertAuthenticatedAs($user);
```

assertVue

Dusk even allows you to make assertions on the state of [Vue component](#) data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/***
 * A basic Vue test example.
 */
public function test_vue(): void
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
            ->assertVue('user.name', 'Taylor', '@profile-component');
    });
}
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

assertVueDoesntContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesntContain($property, $value, $componentSelector = null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Dusk Pages allow you to define expressive actions that may then be performed on a given page via a single method. Pages also allow you to define short-cuts to common selectors for your application or for a single page.

Generating Pages

To generate a page object, execute the `dusk:page` Artisan command. All page objects will be placed in your application's `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be [discussed in more detail below](#).

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**  
 * Get the URL for the page.  
 */  
public function url(): string  
{  
    return '/login';  
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. It is not actually necessary to place anything within this method; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**  
 * Assert that the browser is on the page.  
 */  
public function assert(Browser $browser): void  
{  
    $browser->assertPathIs($this->url());  
}
```

Navigating to Pages

Once a page has been defined, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

Shorthand Selectors

The `elements` method within page classes allows you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array<string, string>
 */
public function elements(): array
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Once the shortcut has been defined, you may use the shorthand selector anywhere you would typically use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array<string, string>
 */
public static function siteElements(): array
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     */
    public function createPlaylist(Browser $browser, string $name): void
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed as the first argument to custom page methods:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, execute the `dusk:component` Artisan command. New components are placed in the `tests/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```

<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     */
    public function selector(): string
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     */
    public function assert(Browser $browser): void
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array<string, string>
     */
    public function elements(): array
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     */
    public function selectDate(Browser $browser, int $year, int $month, int $day): void
    {
        $browser->click('@date-field')
            ->within('@year-list', function (Browser $browser) use ($year) {
                $browser->click($year);
            })
            ->within('@month-list', function (Browser $browser) use ($month) {
                $browser->click($month);
            })
            ->within('@day-list', function (Browser $browser) use ($day) {
                $browser->click($day);
            });
    }
}

```

```

    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```

<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     */
    public function test_basic_example(): void
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                ->within(new DatePicker, function (Browser $browser) {
                    $browser->selectDate(2019, 1, 30);
                })
                ->assertSee('January');
        });
    }
}
```

Continuous Integration



Most Dusk continuous integration configurations expect your Laravel application to be served using the built-in PHP development server on port 8000. Therefore, before continuing, you should ensure that your continuous integration environment has an `APP_URL` environment variable value of `http://127.0.0.1:8000`.

Heroku CI

To run Dusk tests on [Heroku CI](#), add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &' && nohup bash -c 'php artisan serve --no-reload > /dev/null 2>&1 && php artisan dusk' "
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on [Travis CI](#), use the following `.travis.yml` configuration. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost &
  - php artisan serve --no-reload &

script:
  - php artisan dusk
```

GitHub Actions

If you are using [GitHub Actions](#) to run your Dusk tests, you may use the following configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```

name: CI
on: [push]
jobs:

dusk-php:
  runs-on: ubuntu-latest
  env:
    APP_URL: "http://127.0.0.1:8000"
    DB_USERNAME: root
    DB_PASSWORD: root
    MAIL_MAILER: log
  steps:
    - uses: actions/checkout@v4
    - name: Prepare The Environment
      run: cp .env.example .env
    - name: Create Database
      run: |
        sudo systemctl start mysql
        mysql --user="root" --password="root" -e "CREATE DATABASE `my-database`\ character
set UTF8mb4 collate utf8mb4_bin;"
    - name: Install Composer Dependencies
      run: composer install --no-progress --prefer-dist --optimize-autoloader
    - name: Generate Application Key
      run: php artisan key:generate
    - name: Upgrade Chrome Driver
      run: php artisan dusk:chrome-driver --detect
    - name: Start Chrome Driver
      run: ./vendor/laravel/dusk/bin/chromedriver-linux &
    - name: Run Laravel Server
      run: php artisan serve --no-reload &
    - name: Run Dusk Tests
      run: php artisan dusk
    - name: Upload Screenshots
      if: failure()
      uses: actions/upload-artifact@v2
      with:
        name: screenshots
        path: tests/Browser/screenshots
    - name: Upload Console Logs
      if: failure()
      uses: actions/upload-artifact@v2
      with:
        name: console
        path: tests/Browser/console

```

Chipper CI

If you are using [Chipper CI](#) to run your Dusk tests, you may use the following configuration file as a starting point. We will use PHP's built-in server to run Laravel so we can listen for requests:

```
# file .chipperci.yml
version: 1

environment:
  php: 8.2
  node: 16

# Include Chrome in the build environment
services:
  - dusk

# Build all commits
on:
  push:
    branches: .*

pipeline:
  - name: Setup
    cmd: |
      cp -v .env.example .env
      composer install --no-interaction --prefer-dist --optimize-autoloader
      php artisan key:generate

      # Create a dusk env file, ensuring APP_URL uses BUILD_HOST
      cp -v .env .env.dusk.ci
      sed -i "s@APP_URL=.*@APP_URL=http://$BUILD_HOST:8000@g" .env.dusk.ci

  - name: Compile Assets
    cmd: |
      npm ci --no-audit
      npm run build

  - name: Browser Tests
    cmd: |
      php -S [::0]:8000 -t public 2>server.log &
      sleep 2
      php artisan dusk:chrome-driver $CHROME_DRIVER
      php artisan dusk --env=ci
```

To learn more about running Dusk tests on Chipper CI, including how to use databases, consult the [official Chipper CI documentation](#).

Laravel Envoy

- [Introduction](#)
- [Installation](#)
- [Writing Tasks](#)
 - [Defining Tasks](#)
 - [Multiple Servers](#)
 - [Setup](#)
 - [Variables](#)
 - [Stories](#)
 - [Hooks](#)
- [Running Tasks](#)
 - [Confirming Task Execution](#)
- [Notifications](#)
 - [Slack](#)
 - [Discord](#)
 - [Telegram](#)
 - [Microsoft Teams](#)

Introduction

[Laravel Envoy](#) is a tool for executing common tasks you run on your remote servers. Using [Blade](#) style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems. However, Windows support is achievable using [WSL2](#).

Installation

First, install Envoy into your project using the Composer package manager:

```
composer require laravel/envoy --dev
```

Once Envoy has been installed, the Envoy binary will be available in your application's `vendor/bin` directory:

```
php vendor/bin/envoy
```

Writing Tasks

Defining Tasks

Tasks are the basic building block of Envoy. Tasks define the shell commands that should execute on your remote servers when the task is invoked. For example, you might define a task that executes the `php artisan queue:restart` command on all of your application's queue worker servers.

All of your Envoy tasks should be defined in an `Envoy.blade.php` file at the root of your application. Here's an example to get you started:

```
@servers(['web' => ['user@192.168.1.1'], 'workers' => ['user@192.168.1.2']])

@task('restart-queues', ['on' => 'workers'])
    cd /home/user/example.com
    php artisan queue:restart
@endtask
```

As you can see, an array of `@servers` is defined at the top of the file, allowing you to reference these servers via the `on` option of your task declarations. The `@servers` declaration should always be placed on a single line. Within your `@task` declarations, you should place the shell commands that should execute on your servers when the task is invoked.

Local Tasks

You can force a script to run on your local computer by specifying the server's IP address as `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Importing Envoy Tasks

Using the `@import` directive, you may import other Envoy files so their stories and tasks are added to yours. After the files have been imported, you may execute the tasks they contain as if they were defined in your own Envoy file:

```
@import('vendor/package/Envoy.blade.php')
```

Multiple Servers

Envoy allows you to easily run a task across multiple servers. First, add additional servers to your `@servers` declaration. Each server should be assigned a unique name. Once you have defined your additional servers you may list each of the servers in the task's `on` array:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Parallel Execution

By default, tasks will be executed on each server serially. In other words, a task will finish running on the first server before proceeding to execute on the second server. If you would like to run a task across multiple servers in parallel, add the `parallel` option to your task declaration:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Setup

Sometimes, you may need to execute arbitrary PHP code before running your Envoy tasks. You may use the `@setup` directive to define a block of PHP code that should execute before your tasks:

```
@setup
    $now = new DateTime;
@endsetup
```

If you need to require other PHP files before your task is executed, you may use the `@include` directive at the top of your `Envoy.blade.php` file:

```
@include('vendor/autoload.php')

@task('restart-queues')
    # ...
@endtask
```

Variables

If needed, you may pass arguments to Envoy tasks by specifying them on the command line when invoking Envoy:

```
php vendor/bin/envoy run deploy --branch=master
```

You may access the options within your tasks using Blade's "echo" syntax. You may also define Blade `if` statements and loops within your tasks. For example, let's verify the presence of the `$branch` variable before executing the `git pull` command:

```
@servers(['web' => ['user@192.168.1.1']])

@task('deploy', ['on' => 'web'])
    cd /home/user/example.com

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate --force
@endtask
```

Stories

Stories group a set of tasks under a single, convenient name. For instance, a `deploy` story may run the `update-code` and `install-dependencies` tasks by listing the task names within its definition:

```

@servers(['web' => ['user@192.168.1.1']])

@story('deploy')
    update-code
    install-dependencies
@endstory

@task('update-code')
    cd /home/user/example.com
    git pull origin master
@endtask

@task('install-dependencies')
    cd /home/user/example.com
    composer install
@endtask

```

Once the story has been written, you may invoke it in the same way you would invoke a task:

```
php vendor/bin/envoy run deploy
```

Hooks

When tasks and stories run, a number of hooks are executed. The hook types supported by Envoy are `@before`, `@after`, `@error`, `@success`, and `@finished`. All of the code in these hooks is interpreted as PHP and executed locally, not on the remote servers that your tasks interact with.

You may define as many of each of these hooks as you like. They will be executed in the order that they appear in your Envoy script.

`@before`

Before each task execution, all of the `@before` hooks registered in your Envoy script will execute. The `@before` hooks receive the name of the task that will be executed:

```

@before
    if ($task === 'deploy') {
        // ...
    }
@endbefore

```

`@after`

After each task execution, all of the `@after` hooks registered in your Envoy script will execute. The `@after` hooks receive the name of the task that was executed:

```

@after
    if ($task === 'deploy') {
        // ...
    }
@endafter

```

`@error`

After every task failure (exits with a status code greater than `0`), all of the `@error` hooks registered in your Envoy script will execute. The `@error` hooks receive the name of the task that was executed:

```
@error
if ($task === 'deploy') {
    // ...
}
@enderror
```

`@success`

If all tasks have executed without errors, all of the `@success` hooks registered in your Envoy script will execute:

```
@success
// ...
@endsuccess
```

`@finished`

After all tasks have been executed (regardless of exit status), all of the `@finished` hooks will be executed. The `@finished` hooks receive the status code of the completed task, which may be `null` or an `integer` greater than or equal to `0`:

```
@finished
if ($exitCode > 0) {
    // There were errors in one of the tasks...
}
@endfinished
```

Running Tasks

To run a task or story that is defined in your application's `Envoy.blade.php` file, execute Envoy's `run` command, passing the name of the task or story you would like to execute. Envoy will execute the task and display the output from your remote servers as the task is running:

```
php vendor/bin/envoy run deploy
```

Confirming Task Execution

If you would like to be prompted for confirmation before running a given task on your servers, you should add the `confirm` directive to your task declaration. This option is particularly useful for destructive operations:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
cd /home/user/example.com
git pull origin {{ $branch }}
php artisan migrate
@endtask
```

Notifications

Slack

Envoy supports sending notifications to [Slack](#) after each task is executed. The `@slack` directive accepts a Slack hook URL and a channel / user name. You may retrieve your webhook URL by creating an "Incoming WebHooks" integration in your Slack control panel.

You should pass the entire webhook URL as the first argument given to the `@slack` directive. The second argument given to the `@slack` directive should be a channel name (`#channel`) or a user name (`@user`):

```
@finished
@slack('webhook-url', '#bots')
@endfinished
```

By default, Envoy notifications will send a message to the notification channel describing the task that was executed. However, you may overwrite this message with your own custom message by passing a third argument to the `@slack` directive:

```
@finished
@slack('webhook-url', '#bots', 'Hello, Slack.')
@endfinished
```

Discord

Envoy also supports sending notifications to [Discord](#) after each task is executed. The `@discord` directive accepts a Discord hook URL and a message. You may retrieve your webhook URL by creating a "Webhook" in your Server Settings and choosing which channel the webhook should post to. You should pass the entire Webhook URL into the `@discord` directive:

```
@finished
@discord('discord-webhook-url')
@endfinished
```

Telegram

Envoy also supports sending notifications to [Telegram](#) after each task is executed. The `@telegram` directive accepts a Telegram Bot ID and a Chat ID. You may retrieve your Bot ID by creating a new bot using [BotFather](#). You can retrieve a valid Chat ID using [@username_to_id_bot](#). You should pass the entire Bot ID and Chat ID into the `@telegram` directive:

```
@finished
@telegram('bot-id', 'chat-id')
@endfinished
```

Microsoft Teams

Envoy also supports sending notifications to [Microsoft Teams](#) after each task is executed. The `@microsoftTeams` directive accepts a Teams Webhook (required), a message, theme color (success, info, warning, error), and an array of options. You may retrieve your Teams Webhook by creating a new [incoming webhook](#). The Teams API has many other attributes to customize your message box like title, summary, and sections. You can find more information on the [Microsoft Teams documentation](#). You should pass the entire Webhook URL into the `@microsoftTeams` directive:

```
@finished
@microsoftTeams('webhook-url')
@endfinished
```

Laravel Fortify

- [Introduction](#)
 - [What is Fortify?](#)
 - [When Should I Use Fortify?](#)
- [Installation](#)
 - [The Fortify Service Provider](#)
 - [Fortify Features](#)
 - [Disabling Views](#)
- [Authentication](#)
 - [Customizing User Authentication](#)
 - [Customizing the Authentication Pipeline](#)
 - [Customizing Redirects](#)
- [Two Factor Authentication](#)
 - [Enabling Two Factor Authentication](#)
 - [Authenticating With Two Factor Authentication](#)
 - [Disabling Two Factor Authentication](#)
- [Registration](#)
 - [Customizing Registration](#)
- [Password Reset](#)
 - [Requesting a Password Reset Link](#)
 - [Resetting the Password](#)
 - [Customizing Password Resets](#)
- [Email Verification](#)
 - [Protecting Routes](#)
- [Password Confirmation](#)

Introduction

[Laravel Fortify](#) is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more. After installing Fortify, you may run the `route:list` Artisan command to see the routes that Fortify has registered.

Since Fortify does not provide its own user interface, it is meant to be paired with your own user interface which makes requests to the routes it registers. We will discuss exactly how to make requests to these routes in the remainder of this documentation.



*Remember, Fortify is a package that is meant to give you a head start implementing Laravel's authentication features. **You are not required to use it.** You are always free to manually interact with Laravel's authentication services by following the documentation available in the [authentication](#), [password reset](#), and [email verification](#) documentation.*

What is Fortify?

As mentioned previously, Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more.

You are not required to use Fortify in order to use Laravel's authentication features. You are always free to manually interact with Laravel's authentication services by following the documentation available in the [authentication](#), [password reset](#), and [email verification](#) documentation.

If you are new to Laravel, you may wish to explore the [Laravel Breeze](#) application starter kit before attempting to use Laravel Fortify. Laravel Breeze provides an authentication scaffolding for your application that includes a user interface built with [Tailwind CSS](#). Unlike Fortify, Breeze publishes its routes and controllers directly into your application. This allows you to study and get comfortable with Laravel's authentication features before allowing Laravel Fortify to implement these features for you.

Laravel Fortify essentially takes the routes and controllers of Laravel Breeze and offers them as a package that does not include a user interface. This allows you to still quickly scaffold the backend implementation of your application's authentication layer without being tied to any particular frontend opinions.

When Should I Use Fortify?

You may be wondering when it is appropriate to use Laravel Fortify. First, if you are using one of Laravel's [application starter kits](#), you do not need to install Laravel Fortify since all of Laravel's application starter kits already provide a full authentication implementation.

If you are not using an application starter kit and your application needs authentication features, you have two options: manually implement your application's authentication features or use Laravel Fortify to provide the backend implementation of these features.

If you choose to install Fortify, your user interface will make requests to Fortify's authentication routes that are detailed in this documentation in order to authenticate and register users.

If you choose to manually interact with Laravel's authentication services instead of using Fortify, you may do so by following the documentation available in the [authentication](#), [password reset](#), and [email verification](#) documentation.

Laravel Fortify and Laravel Sanctum

Some developers become confused regarding the difference between [Laravel Sanctum](#) and Laravel Fortify. Because the two packages solve two different but related problems, Laravel Fortify and Laravel Sanctum are not mutually exclusive or competing packages.

Laravel Sanctum is only concerned with managing API tokens and authenticating existing users using session cookies or tokens. Sanctum does not provide any routes that handle user registration, password reset, etc.

If you are attempting to manually build the authentication layer for an application that offers an API or serves as the backend for a single-page application, it is entirely possible that you will utilize both Laravel Fortify (for user registration, password reset, etc.) and Laravel Sanctum (API token management, session authentication).

Installation

To get started, install Fortify using the Composer package manager:

```
composer require laravel/fortify
```

Next, publish Fortify's resources using the `vendor:publish` command:

```
php artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProvider"
```

This command will publish Fortify's actions to your `app/Actions` directory, which will be created if it does not exist. In addition, the `FortifyServiceProvider` configuration file, and all necessary database migrations will be published.

Next, you should migrate your database:

```
php artisan migrate
```

The Fortify Service Provider

The `vendor:publish` command discussed above will also publish the `App\Providers\FortifyServiceProvider` class. You should ensure this class is registered within the `providers` array of your application's `config/app.php` configuration file.

The Fortify service provider registers the actions that Fortify published and instructs Fortify to use them when their respective tasks are executed by Fortify.

Fortify Features

The `fortify` configuration file contains a `features` configuration array. This array defines which backend routes / features Fortify will expose by default. If you are not using Fortify in combination with [Laravel Jetstream](#), we recommend that you only enable the following features, which are the basic authentication features provided by most Laravel applications:

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
],
```

Disabling Views

By default, Fortify defines routes that are intended to return views, such as a login screen or registration screen. However, if you are building a JavaScript driven single-page application, you may not need these routes. For that reason, you may disable these routes entirely by setting the `views` configuration value within your application's `config/fortify.php` configuration file to `false`:

```
'views' => false,
```

Disabling Views and Password Reset

If you choose to disable Fortify's views and you will be implementing password reset features for your application, you should still define a route named `password.reset` that is responsible for displaying your application's "reset password" view. This is necessary because Laravel's `Illuminate\Auth\Notifications\ResetPassword` notification will generate the password reset URL via the `password.reset` named route.

Authentication

To get started, we need to instruct Fortify how to return our "login" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an [application starter kit](#).

All of the authentication view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class. Fortify will take care of defining the `/login` route that returns this view:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::loginView(function () {
        return view('auth.login');
    });

    // ...
}
```

Your login template should include a form that makes a POST request to `/login`. The `/login` endpoint expects a string `email` / `username` and a `password`. The name of the email / username field should match the `username` value within the `config/fortify.php` configuration file. In addition, a boolean `remember` field may be provided to indicate that the user would like to use the "remember me" functionality provided by Laravel.

If the login attempt is successful, Fortify will redirect you to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the login screen and the validation errors will be available to you via the shared `$errors` [Blade template variable](#). Or, in the case of an XHR request, the validation errors will be returned with the 422 HTTP response.

Customizing User Authentication

Fortify will automatically retrieve and authenticate the user based on the provided credentials and the authentication guard that is configured for your application. However, you may sometimes wish to have full customization over how login credentials are authenticated and users are retrieved. Thankfully, Fortify allows you to easily accomplish this using the `Fortify::authenticateUsing` method.

This method accepts a closure which receives the incoming HTTP request. The closure is responsible for validating the login credentials attached to the request and returning the associated user instance. If the credentials are invalid or no user can be found, `null` or `false` should be returned by the closure. Typically, this method should be called from the `boot` method of your `FortifyServiceProvider`:

```

use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::authenticateUsing(function (Request $request) {
        $user = User::where('email', $request->email)->first();

        if ($user &&
            Hash::check($request->password, $user->password)) {
            return $user;
        }
    });

    // ...
}

```

Authentication Guard

You may customize the authentication guard used by Fortify within your application's `fortify` configuration file.

However, you should ensure that the configured guard is an implementation of

`Illuminate\Contracts\Auth\StatefulGuard`. If you are attempting to use Laravel Fortify to authenticate an SPA, you should use Laravel's default `web` guard in combination with [Laravel Sanctum](#).

Customizing the Authentication Pipeline

Laravel Fortify authenticates login requests through a pipeline of invokable classes. If you would like, you may define a custom pipeline of classes that login requests should be piped through. Each class should have an `__invoke` method which receives the incoming `Illuminate\Http\Request` instance and, like `middleware`, a `$next` variable that is invoked in order to pass the request to the next class in the pipeline.

To define your custom pipeline, you may use the `Fortify::authenticateThrough` method. This method accepts a closure which should return the array of classes to pipe the login request through. Typically, this method should be called from the `boot` method of your `App\Providers\FortifyServiceProvider` class.

The example below contains the default pipeline definition that you may use as a starting point when making your own modifications:

```

use Laravel\Fortify\Actions\AttemptToAuthenticate;
use Laravel\Fortify\Actions\EnsureLoginIsNotThrottled;
use Laravel\Fortify\Actions\PrepareAuthenticatedSession;
use Laravel\Fortify\Actions\RedirectIfTwoFactorAuthenticatable;
use Laravel\Fortify\Fortify;
use Illuminate\Http\Request;

Fortify::authenticateThrough(function (Request $request) {
    return array_filter([
        config('fortify.limiters.login') ? null : EnsureLoginIsNotThrottled::class,
        Features::enabled(Features::twoFactorAuthentication()) ? RedirectIfTwoFactorAuthenticatable::class : null,
        AttemptToAuthenticate::class,
        PrepareAuthenticatedSession::class,
    ]);
});

```

Customizing Redirects

If the login attempt is successful, Fortify will redirect you to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned. After a user logs out of the application, the user will be redirected to the `/` URI.

If you need advanced customization of this behavior, you may bind implementations of the `LoginResponse` and `LogoutResponse` contracts into the Laravel service container. Typically, this should be done within the `register` method of your application's `App\Providers\FortifyServiceProvider` class:

```

use Laravel\Fortify\Contracts\LogoutResponse;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->instance(LogoutResponse::class, new class implements LogoutResponse {
        public function toResponse($request)
        {
            return redirect('/');
        }
    });
}

```

Two Factor Authentication

When Fortify's two factor authentication feature is enabled, the user is required to input a six digit numeric token during the authentication process. This token is generated using a time-based one-time password (TOTP) that can be retrieved from any TOTP compatible mobile authentication application such as Google Authenticator.

Before getting started, you should first ensure that your application's `App\Models\User` model uses the `Laravel\Fortify\TwoFactorAuthenticatable` trait:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Fortify\TwoFactorAuthenticatable;

class User extends Authenticatable
{
    use Notifiable, TwoFactorAuthenticatable;
}
```

Next, you should build a screen within your application where users can manage their two factor authentication settings. This screen should allow the user to enable and disable two factor authentication, as well as regenerate their two factor authentication recovery codes.

By default, the `[features]` array of the `[fortify]` configuration file instructs Fortify's two factor authentication settings to require password confirmation before modification. Therefore, your application should implement Fortify's [password confirmation](#) feature before continuing.

Enabling Two Factor Authentication

To begin enabling two factor authentication, your application should make a POST request to the `/user/two-factor-authentication` endpoint defined by Fortify. If the request is successful, the user will be redirected back to the previous URL and the `[status]` session variable will be set to `[two-factor-authentication-enabled]`. You may detect this `[status]` session variable within your templates to display the appropriate success message. If the request was an XHR request, `200` HTTP response will be returned.

After choosing to enable two factor authentication, the user must still "confirm" their two factor authentication configuration by providing a valid two factor authentication code. So, your "success" message should instruct the user that two factor authentication confirmation is still required:

```
@if (session('status') == 'two-factor-authentication-enabled')
    <div class="mb-4 font-medium text-sm">
        Please finish configuring two factor authentication below.
    </div>
@endif
```

Next, you should display the two factor authentication QR code for the user to scan into their authenticator application. If you are using Blade to render your application's frontend, you may retrieve the QR code SVG using the `[twoFactorQrCodeSvg]` method available on the user instance:

```
$request->user()->twoFactorQrCodeSvg();
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-qrcode` endpoint to retrieve the user's two factor authentication QR code. This endpoint will return a JSON object containing an `svg` key.

Confirming Two Factor Authentication

In addition to displaying the user's two factor authentication QR code, you should provide a text input where the user can supply a valid authentication code to "confirm" their two factor authentication configuration. This code should be provided to the Laravel application via a POST request to the `/user/confirmed-two-factor-authentication` endpoint defined by Fortify.

If the request is successful, the user will be redirected back to the previous URL and the `status` session variable will be set to `two-factor-authentication-confirmed`:

```
@if (session('status') == 'two-factor-authentication-confirmed')
    <div class="mb-4 font-medium text-sm">
        Two factor authentication confirmed and enabled successfully.
    </div>
@endif
```

If the request to the two factor authentication confirmation endpoint was made via an XHR request, a `200` HTTP response will be returned.

Displaying the Recovery Codes

You should also display the user's two factor recovery codes. These recovery codes allow the user to authenticate if they lose access to their mobile device. If you are using Blade to render your application's frontend, you may access the recovery codes via the authenticated user instance:

```
(array) $request->user()->recoveryCodes()
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-recovery-codes` endpoint. This endpoint will return a JSON array containing the user's recovery codes.

To regenerate the user's recovery codes, your application should make a POST request to the `/user/two-factor-recovery-codes` endpoint.

Authenticating With Two Factor Authentication

During the authentication process, Fortify will automatically redirect the user to your application's two factor authentication challenge screen. However, if your application is making an XHR login request, the JSON response returned after a successful authentication attempt will contain a JSON object that has a `two_factor` boolean property. You should inspect this value to know whether you should redirect to your application's two factor authentication challenge screen.

To begin implementing two factor authentication functionality, we need to instruct Fortify how to return our two factor authentication challenge view. All of Fortify's authentication view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::twoFactorChallengeView(function () {
        return view('auth.two-factor-challenge');
    });

    // ...
}
```

Fortify will take care of defining the `/two-factor-challenge` route that returns this view. Your `two-factor-challenge` template should include a form that makes a POST request to the `/two-factor-challenge` endpoint. The `/two-factor-challenge` action expects a `code` field that contains a valid TOTP token or a `recovery_code` field that contains one of the user's recovery codes.

If the login attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 204 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the two factor challenge screen and the validation errors will be available to you via the shared `$errors` [Blade template variable](#). Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Disabling Two Factor Authentication

To disable two factor authentication, your application should make a DELETE request to the `/user/two-factor-authentication` endpoint. Remember, Fortify's two factor authentication endpoints require [password confirmation](#) prior to being called.

Registration

To begin implementing our application's registration functionality, we need to instruct Fortify how to return our "register" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an [application starter kit](#).

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::registerView(function () {
        return view('auth.register');
    });

    // ...
}
```

Fortify will take care of defining the `/register` route that returns this view. Your `register` template should include a form that makes a POST request to the `/register` endpoint defined by Fortify.

The `/register` endpoint expects a string `name`, string email address / username, `password`, and `password_confirmation` fields. The name of the email / username field should match the `username` configuration value defined within your application's `fortify` configuration file.

If the registration attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the request was an XHR request, a 201 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the registration screen and the validation errors will be available to you via the shared `$errors` [Blade template variable](#). Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Registration

The user validation and creation process may be customized by modifying the `App\Actions\Fortify\CreateNewUser` action that was generated when you installed Laravel Fortify.

Password Reset

Requesting a Password Reset Link

To begin implementing our application's password reset functionality, we need to instruct Fortify how to return our "forgot password" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an [application starter kit](#).

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::requestPasswordResetLinkView(function () {
        return view('auth.forgot-password');
    });

    // ...
}
```

Fortify will take care of defining the `/forgot-password` endpoint that returns this view. Your `forgot-password` template should include a form that makes a POST request to the `/forgot-password` endpoint.

The `/forgot-password` endpoint expects a string `email` field. The name of this field / database column should match the `email` configuration value within your application's `fortify` configuration file.

Handling the Password Reset Link Request Response

If the password reset link request was successful, Fortify will redirect the user back to the `/forgot-password` endpoint and send an email to the user with a secure link they can use to reset their password. If the request was an XHR request, a 200 HTTP response will be returned.

After being redirected back to the `/forgot-password` endpoint after a successful request, the `status` session variable may be used to display the status of the password reset link request attempt.

The value of the `$status` session variable will match one of the translation strings defined within your application's `passwords` language file. If you would like to customize this value and have not published Laravel's language files, you may do so via the `lang:publish` Artisan command:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was not successful, the user will be redirected back to the request password reset link screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Resetting the Password

To finish implementing our application's password reset functionality, we need to instruct Fortify how to return our "reset password" view.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;
use Illuminate\Http\Request;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::resetPasswordView(function (Request $request) {
        return view('auth.reset-password', ['request' => $request]);
    });

    // ...
}
```

Fortify will take care of defining the route to display this view. Your `reset-password` template should include a form that makes a POST request to `/reset-password`.

The `/reset-password` endpoint expects a string `email` field, a `password` field, a `password_confirmation` field, and a hidden field named `token` that contains the value of `request()->route('token')`. The name of the "email" field / database column should match the `email` configuration value defined within your application's `fortify` configuration file.

Handling the Password Reset Response

If the password reset request was successful, Fortify will redirect back to the `/login` route so that the user can log in with their new password. In addition, a `status` session variable will be set so that you may display the successful status of the reset on your login screen:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the reset password screen and the validation errors will be available to you via the shared `$errors` [Blade template variable](#). Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Password Resets

The password reset process may be customized by modifying the `App\Actions\ResetUserPassword` action that was generated when you installed Laravel Fortify.

Email Verification

After registration, you may wish for users to verify their email address before they continue accessing your application. To get started, ensure the `emailVerification` feature is enabled in your `fortify` configuration file's `features` array. Next, you should ensure that your `App\Models\User` class implements the `Illuminate\Contracts\Auth\MustVerifyEmail` interface.

Once these two setup steps have been completed, newly registered users will receive an email prompting them to verify their email address ownership. However, we need to inform Fortify how to display the email verification screen which informs the user that they need to go click the verification link in the email.

All of Fortify's view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::verifyEmailView(function () {
        return view('auth.verify-email');
    });

    // ...
}
```

Fortify will take care of defining the route that displays this view when a user is redirected to the `/email/verify` endpoint by Laravel's built-in `verified` middleware.

Your `verify-email` template should include an informational message instructing the user to click the email verification link that was sent to their email address.

Resending Email Verification Links

If you wish, you may add a button to your application's `verify-email` template that triggers a POST request to the `/email/verification-notification` endpoint. When this endpoint receives a request, a new verification email link will be emailed to the user, allowing the user to get a new verification link if the previous one was accidentally deleted or lost.

If the request to resend the verification link email was successful, Fortify will redirect the user back to the `/email/verify` endpoint with a `status` session variable, allowing you to display an informational message to the user informing them the operation was successful. If the request was an XHR request, a 202 HTTP response will be returned:

```
@if (session('status') == 'verification-link-sent')
    <div class="mb-4 font-medium text-sm text-green-600">
        A new email verification link has been emailed to you!
    </div>
@endif
```

Protecting Routes

To specify that a route or group of routes requires that the user has verified their email address, you should attach Laravel's built-in `verified` middleware to the route. This middleware is registered within your application's `App\Http\Kernel` class:

```
Route::get('/dashboard', function () {
    // ...
})->middleware(['verified']);
```

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed. Typically, these routes are protected by Laravel's built-in `password.confirm` middleware.

To begin implementing password confirmation functionality, we need to instruct Fortify how to return our application's "password confirmation" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an [application starter kit](#).

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::confirmPasswordView(function () {
        return view('auth.confirm-password');
    });

    // ...
}
```

Fortify will take care of defining the `/user/confirm-password` endpoint that returns this view. Your `confirm-password` template should include a form that makes a POST request to the `/user/confirm-password` endpoint. The `/user/confirm-password` endpoint expects a `password` field that contains the user's current password.

If the password matches the user's current password, Fortify will redirect the user to the route they were attempting to access. If the request was an XHR request, a 201 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the confirm password screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Laravel Folio

- [Introduction](#)
- [Installation](#)
 - [Page Paths / URIs](#)
 - [Subdomain Routing](#)
- [Creating Routes](#)
 - [Nested Routes](#)
 - [Index Routes](#)
- [Route Parameters](#)
- [Route Model Binding](#)
 - [Soft Deleted Models](#)
- [Render Hooks](#)
- [Named Routes](#)
- [Middleware](#)
- [Route Caching](#)

Introduction

[Laravel Folio](#) is a powerful page based router designed to simplify routing in Laravel applications. With Laravel Folio, generating a route becomes as effortless as creating a Blade template within your application's `resources/views/pages` directory.

For example, to create a page that is accessible at the `/greeting` URL, just create a `greeting.blade.php` file in your application's `resources/views/pages` directory:

```
<div>
    Hello World
</div>
```

Installation

To get started, install Folio into your project using the Composer package manager:

```
composer require laravel/folio
```

After installing Folio, you may execute the `folio:install` Artisan command, which will install Folio's service provider into your application. This service provider registers the directory where Folio will search for routes / pages:

```
php artisan folio:install
```

Page Paths / URIs

By default, Folio serves pages from your application's `resources/views/pages` directory, but you may customize these directories in your Folio service provider's `boot` method.

For example, sometimes it may be convenient to specify multiple Folio paths in the same Laravel application. You may wish to have a separate directory of Folio pages for your application's "admin" area, while using another directory for the rest of your application's pages.

You may accomplish this using the `Folio::path` and `Folio::uri` methods. The `path` method registers a directory that Folio will scan for pages when routing incoming HTTP requests, while the `uri` method specifies the "base URI" for

that directory of pages:

```
use Laravel\Folio\Folio;

Folio::path(resource_path('views/pages/guest'))->uri('/');

Folio::path(resource_path('views/pages/admin'))
    ->uri('/admin')
    ->middleware([
        '*' => [
            'auth',
            'verified',
        ],
    ],
);
```

Subdomain Routing

You may also route to pages based on the incoming request's subdomain. For example, you may wish to route requests from `admin.example.com` to a different page directory than the rest of your Folio pages. You may accomplish this by invoking the `domain` method after invoking the `Folio::path` method:

```
use Laravel\Folio\Folio;

Folio::domain('admin.example.com')
    ->path(resource_path('views/pages/admin'));
```

The `domain` method also allows you to capture parts of the domain or subdomain as parameters. These parameters will be injected into your page template:

```
use Laravel\Folio\Folio;

Folio::domain('{account}.example.com')
    ->path(resource_path('views/pages/admin'));
```

Creating Routes

You may create a Folio route by placing a Blade template in any of your Folio mounted directories. By default, Folio mounts the `resources/views/pages` directory, but you may customize these directories in your Folio service provider's `boot` method.

Once a Blade template has been placed in a Folio mounted directory, you may immediately access it via your browser. For example, a page placed in `pages/schedule.blade.php` may be accessed in your browser at `http://example.com/schedule`.

To quickly view a list of all of your Folio pages / routes, you may invoke the `folio:list` Artisan command:

```
php artisan folio:list
```

Nested Routes

You may create a nested route by creating one or more directories within one of Folio's directories. For instance, to create a page that is accessible via `/user/profile`, create a `profile.blade.php` template within the `pages/user`

directory:

```
php artisan make:folio user/profile

# pages/user/profile.blade.php → /user/profile
```

Index Routes

Sometimes, you may wish to make a given page the "index" of a directory. By placing an `index.blade.php` template within a Folio directory, any requests to the root of that directory will be routed to that page:

```
php artisan make:folio index
# pages/index.blade.php → /

php artisan make:folio users/index
# pages/users/index.blade.php → /users
```

Route Parameters

Often, you will need to have segments of the incoming request's URL injected into your page so that you can interact with them. For example, you may need to access the "ID" of the user whose profile is being displayed. To accomplish this, you may encapsulate a segment of the page's filename in square brackets:

```
php artisan make:folio "users/[id]"

# pages/users/[id].blade.php → /users/1
```

Captured segments can be accessed as variables within your Blade template:

```
<div>
    User {{ $id }}
</div>
```

To capture multiple segments, you can prefix the encapsulated segment with three dots `...`:

```
php artisan make:folio "users/...ids"

# pages/users/...ids.blade.php → /users/1/2/3
```

When capturing multiple segments, the captured segments will be injected into the page as an array:

```
<ul>
    @foreach ($ids as $id)
        <li>User {{ $id }}</li>
    @endforeach
</ul>
```

Route Model Binding

If a wildcard segment of your page template's filename corresponds one of your application's Eloquent models, Folio will automatically take advantage of Laravel's route model binding capabilities and attempt to inject the resolved model instance into your page:

```
php artisan make:folio "users/[User]"  
# pages/users/[User].blade.php → /users/1
```

Captured models can be accessed as variables within your Blade template. The model's variable name will be converted to "camel case":

```
<div>  
    User {{ $user->id }}  
</div>
```

Customizing the Key

Sometimes you may wish to resolve bound Eloquent models using a column other than `[id]`. To do so, you may specify the column in the page's filename. For example, a page with the filename `[Post:slug].blade.php` will attempt to resolve the bound model via the `slug` column instead of the `[id]` column.

On Windows, you should use `-` to separate the model name from the key: `[Post-slug].blade.php`.

Model Location

By default, Folio will search for your model within your application's `app/Models` directory. However, if needed, you may specify the fully-qualified model class name in your template's filename:

```
php artisan make:folio "users/ [.App.Models.User]"  
# pages/users/ [.App.Models.User].blade.php → /users/1
```

Soft Deleted Models

By default, models that have been soft deleted are not retrieved when resolving implicit model bindings. However, if you wish, you can instruct Folio to retrieve soft deleted models by invoking the `withTrashed` function within the page's template:

```
<?php  
  
use function Laravel\Folio\{withTrashed};  
  
withTrashed();  
  
?>  
  
<div>  
    User {{ $user->id }}  
</div>
```

Render Hooks

By default, Folio will return the content of the page's Blade template as the response to the incoming request. However, you may customize the response by invoking the `render` function within the page's template.

The `render` function accepts a closure which will receive the `View` instance being rendered by Folio, allowing you to add additional data to the view or customize the entire response. In addition to receiving the `View` instance, any additional route parameters or model bindings will also be provided to the `render` closure:

```
<?php

use App\Models\Post;
use Illuminate\Support\Facades\Auth;
use Illuminate\View\View;

use function Laravel\Folio\render;

render(function (View $view, Post $post) {
    if (! Auth::user()->can('view', $post)) {
        return response('Unauthorized', 403);
    }

    return $view->with('photos', $post->author->photos);
}); ?>

<div>
    {{ $post->content }}
</div>

<div>
    This author has also taken {{ count($photos) }} photos.
</div>
```

Named Routes

You may specify a name for a given page's route using the `name` function:

```
<?php

use function Laravel\Folio\name;

name('users.index');
```

Just like Laravel's named routes, you may use the `route` function to generate URLs to Folio pages that have been assigned a name:

```
<a href="{{ route('users.index') }}">
    All Users
</a>
```

If the page has parameters, you may simply pass their values to the `route` function:

```
route('users.show', ['user' => $user]);
```

Middleware

You can apply middleware to a specific page by invoking the `middleware` function within the page's template:

```
<?php

use function Laravel\Folio\{middleware};

middleware(['auth', 'verified']);

?>

<div>
    Dashboard
</div>
```

Or, to assign middleware to a group of pages, you may chain the `middleware` method after invoking the `Folio::path` method.

To specify which pages the middleware should be applied to, the array of middleware may be keyed using the corresponding URL patterns of the pages they should be applied to. The `*` character may be utilized as a wildcard character:

```
use Laravel\Folio\Folio;

Folio::path(resource_path('views/pages'))->middleware([
    'admin/*' => [
        'auth',
        'verified',

        // ...
    ],
]);
```

You may include closures in the array of middleware to define inline, anonymous middleware:

```
use Closure;
use Illuminate\Http\Request;
use Laravel\Folio\Folio;

Folio::path(resource_path('views/pages'))->middleware([
    'admin/*' => [
        'auth',
        'verified',

        function (Request $request, Closure $next) {
            // ...

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

Route Caching

When using Folio, you should always take advantage of [Laravel's route caching capabilities](#). Folio listens for the `route:cache` Artisan command to ensure that Folio page definitions and route names are properly cached for maximum performance.

Laravel Homestead

- [Introduction](#)
- [Installation and Setup](#)
 - [First Steps](#)
 - [Configuring Homestead](#)
 - [Configuring Nginx Sites](#)
 - [Configuring Services](#)
 - [Launching the Vagrant Box](#)
 - [Per Project Installation](#)
 - [Installing Optional Features](#)
 - [Aliases](#)
- [Updating Homestead](#)
- [Daily Usage](#)
 - [Connecting via SSH](#)
 - [Adding Additional Sites](#)
 - [Environment Variables](#)
 - [Ports](#)
 - [PHP Versions](#)
 - [Connecting to Databases](#)
 - [Database Backups](#)
 - [Configuring Cron Schedules](#)
 - [Configuring Mailpit](#)
 - [Configuring Minio](#)
 - [Laravel Dusk](#)
 - [Sharing Your Environment](#)
- [Debugging and Profiling](#)
 - [Debugging Web Requests With Xdebug](#)
 - [Debugging CLI Applications](#)
 - [Profiling Applications With Blackfire](#)
- [Network Interfaces](#)
- [Extending Homestead](#)
- [Provider Specific Settings](#)
 - [VirtualBox](#)

Introduction

Laravel strives to make the entire PHP development experience delightful, including your local development environment. [Laravel Homestead](#) is an official, pre-packaged Vagrant box that provides you a wonderful development environment without requiring you to install PHP, a web server, or any other server software on your local machine.

[Vagrant](#) provides a simple, elegant way to manage and provision Virtual Machines. Vagrant boxes are completely disposable. If something goes wrong, you can destroy and re-create the box in minutes!

Homestead runs on any Windows, macOS, or Linux system and includes Nginx, PHP, MySQL, PostgreSQL, Redis, Memcached, Node, and all of the other software you need to develop amazing Laravel applications.



If you are using Windows, you may need to enable hardware virtualization (VT-x). It can usually be enabled via your BIOS. If you are using Hyper-V on a UEFI system you may additionally need to disable Hyper-V in order to access VT-x.

Included Software

- Ubuntu 22.04
- Git
- PHP 8.3
- PHP 8.2
- PHP 8.1
- PHP 8.0
- PHP 7.4
- PHP 7.3
- PHP 7.2
- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- MySQL 8.0
- Imrn
- Sqlite3
- PostgreSQL 15
- Composer
- Docker
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailpit
- avahi
- ngrok
- Xdebug
- XHProf / Tideways / XHGui
- wp-cli

Optional Software

- Apache
- Blackfire
- Cassandra
- Chronograf
- CouchDB
- Crystal & Lucky Framework
- Elasticsearch
- EventStoreDB
- Flyway
- Gearman
- Go
- Grafana
- InfluxDB
- Logstash
- MariaDB
- Meilisearch
- MinIO
- MongoDB
- Neo4j
- Oh My Zsh
- Open Resty
- PM2
- Python
- R
- RabbitMQ
- Rust
- RVM (Ruby Version Manager)
- Solr
- TimescaleDB
- Trader (PHP extension)
- Webdriver & Laravel Dusk Utilities

Installation and Setup

First Steps

Before launching your Homestead environment, you must install [Vagrant](#) as well as one of the following supported providers:

- [VirtualBox 6.1.x](#)
- [Parallels](#)

All of these software packages provide easy-to-use visual installers for all popular operating systems.

To use the Parallels provider, you will need to install [Parallels Vagrant plug-in](#). It is free of charge.

Installing Homestead

You may install Homestead by cloning the Homestead repository onto your host machine. Consider cloning the repository into a `Homestead` folder within your "home" directory, as the Homestead virtual machine will serve as the host to all of your Laravel applications. Throughout this documentation, we will refer to this directory as your "Homestead directory":

```
git clone https://github.com/laravel/homestead.git ~/Homestead
```

After cloning the Laravel Homestead repository, you should checkout the `release` branch. This branch always contains the latest stable release of Homestead:

```
cd ~/Homestead
```

```
git checkout release
```

Next, execute the `bash init.sh` command from the Homestead directory to create the `Homestead.yaml` configuration file. The `Homestead.yaml` file is where you will configure all of the settings for your Homestead installation. This file will be placed in the Homestead directory:

```
# macOS / Linux...
bash init.sh

# Windows...
init.bat
```

Configuring Homestead

Setting Your Provider

The `provider` key in your `Homestead.yaml` file indicates which Vagrant provider should be used: `virtualbox` or `parallels`:

```
provider: virtualbox
```



If you are using Apple Silicon the Parallels provider is required.

Configuring Shared Folders

The `folders` property of the `Homestead.yaml` file lists all of the folders you wish to share with your Homestead environment. As files within these folders are changed, they will be kept in sync between your local machine and the Homestead virtual environment. You may configure as many shared folders as necessary:

```
folders:
  - map: ~/code/project1
    to: /home/vagrant/project1
```



Windows users should not use the `~/` path syntax and instead should use the full path to their project, such as `C:\Users\user\Code\project1`.

You should always map individual applications to their own folder mapping instead of mapping a single large directory that contains all of your applications. When you map a folder, the virtual machine must keep track of all disk IO for every file in the folder. You may experience reduced performance if you have a large number of files in a folder:

```
folders:
- map: ~/code/project1
  to: /home/vagrant/project1
- map: ~/code/project2
  to: /home/vagrant/project2
```



You should never mount `.` (the current directory) when using Homestead. This causes Vagrant to not map the current folder to `/vagrant` and will break optional features and cause unexpected results while provisioning.

To enable NFS, you may add a `type` option to your folder mapping:

```
folders:
- map: ~/code/project1
  to: /home/vagrant/project1
  type: "nfs"
```



When using NFS on Windows, you should consider installing the [vagrant-winnfsd](#) plug-in. This plug-in will maintain the correct user / group permissions for files and directories within the Homestead virtual machine.

You may also pass any options supported by Vagrant's Synced Folders by listing them under the `options` key:

```
folders:
- map: ~/code/project1
  to: /home/vagrant/project1
  type: "rsync"
  options:
    rsync__args: ["--verbose", "--archive", "--delete", "-zz"]
    rsync__exclude: ["node_modules"]
```

Configuring Nginx Sites

Not familiar with Nginx? No problem. Your `Homestead.yaml` file's `sites` property allows you to easily map a "domain" to a folder on your Homestead environment. A sample site configuration is included in the `Homestead.yaml` file. Again, you may add as many sites to your Homestead environment as necessary. Homestead can serve as a convenient, virtualized environment for every Laravel application you are working on:

```
sites:
- map: homestead.test
  to: /home/vagrant/project1/public
```

If you change the `sites` property after provisioning the Homestead virtual machine, you should execute the `vagrant reload --provision` command in your terminal to update the Nginx configuration on the virtual machine.



Homestead scripts are built to be as idempotent as possible. However, if you are experiencing issues while provisioning you should destroy and rebuild the machine by executing the `vagrant destroy && vagrant up` command.

Hostname Resolution

Homestead publishes hostnames using `mDNS` for automatic host resolution. If you set `hostname: homestead` in your `Homestead.yaml` file, the host will be available at `homestead.local`. macOS, iOS, and Linux desktop distributions include `mDNS` support by default. If you are using Windows, you must install [Bonjour Print Services for Windows](#).

Using automatic hostnames works best for [per project installations](#) of Homestead. If you host multiple sites on a single Homestead instance, you may add the "domains" for your web sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead sites into your Homestead virtual machine. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

```
192.168.56.56 homestead.test
```

Make sure the IP address listed is the one set in your `Homestead.yaml` file. Once you have added the domain to your `hosts` file and launched the Vagrant box you will be able to access the site via your web browser:

```
http://homestead.test
```

Configuring Services

Homestead starts several services by default; however, you may customize which services are enabled or disabled during provisioning. For example, you may enable PostgreSQL and disable MySQL by modifying the `services` option within your `Homestead.yaml` file:

```
services:
  - enabled:
    - "postgresql"
  - disabled:
    - "mysql"
```

The specified services will be started or stopped based on their order in the `enabled` and `disabled` directives.

Launching the Vagrant Box

Once you have edited the `Homestead.yaml` to your liking, run the `vagrant up` command from your Homestead directory. Vagrant will boot the virtual machine and automatically configure your shared folders and Nginx sites.

To destroy the machine, you may use the `vagrant destroy` command.

Per Project Installation

Instead of installing Homestead globally and sharing the same Homestead virtual machine across all of your projects, you may instead configure a Homestead instance for each project you manage. Installing Homestead per project may be beneficial if you wish to ship a `Vagrantfile` with your project, allowing others working on the project to `vagrant up` immediately after cloning the project's repository.

You may install Homestead into your project using the Composer package manager:

```
composer require laravel/homestead --dev
```

Once Homestead has been installed, invoke Homestead's `make` command to generate the `Vagrantfile` and `Homestead.yaml` file for your project. These files will be placed in the root of your project. The `make` command will automatically configure the `sites` and `folders` directives in the `Homestead.yaml` file:

```
# macOS / Linux...
php vendor/bin/homestead make

# Windows...
vendor\bin\homestead make
```

Next, run the `vagrant up` command in your terminal and access your project at `http://homestead.test` in your browser. Remember, you will still need to add an `/etc/hosts` file entry for `homestead.test` or the domain of your choice if you are not using automatic [hostname resolution](#).

Installing Optional Features

Optional software is installed using the `features` option within your `Homestead.yaml` file. Most features can be enabled or disabled with a boolean value, while some features allow multiple configuration options:

```

features:
  - blackfire:
      server_id: "server_id"
      server_token: "server_value"
      client_id: "client_id"
      client_token: "client_value"
  - cassandra: true
  - chronograf: true
  - couchdb: true
  - crystal: true
  - dragonflydb: true
  - elasticsearch:
      version: 7.9.0
  - eventstore: true
      version: 21.2.0
  - flyway: true
  - gearman: true
  - golang: true
  - grafana: true
  - influxdb: true
  - logstash: true
  - mariadb: true
  - meilisearch: true
  - minio: true
  - mongodb: true
  - neo4j: true
  - ohmyzsh: true
  - openresty: true
  - pm2: true
  - python: true
  - r-base: true
  - rabbitmq: true
  - rustc: true
  - rvm: true
  - solr: true
  - timescaledb: true
  - trader: true
  - webdriver: true

```

Elasticsearch

You may specify a supported version of Elasticsearch, which must be an exact version number (major.minor.patch). The default installation will create a cluster named 'homestead'. You should never give Elasticsearch more than half of the operating system's memory, so make sure your Homestead virtual machine has at least twice the Elasticsearch allocation.



Check out the [Elasticsearch documentation](#) to learn how to customize your configuration.

MariaDB

Enabling MariaDB will remove MySQL and install MariaDB. MariaDB typically serves as a drop-in replacement for MySQL, so you should still use the `mysql` database driver in your application's database configuration.

MongoDB

The default MongoDB installation will set the database username to `homestead` and the corresponding password to `secret`.

Neo4j

The default Neo4j installation will set the database username to `homestead` and the corresponding password to `secret`. To access the Neo4j browser, visit `http://homestead.test:7474` via your web browser. The ports `7687` (Bolt), `7474` (HTTP), and `7473` (HTTPS) are ready to serve requests from the Neo4j client.

Aliases

You may add Bash aliases to your Homestead virtual machine by modifying the `aliases` file within your Homestead directory:

```
alias c='clear'
alias ..='cd ..'
```

After you have updated the `aliases` file, you should re-provision the Homestead virtual machine using the `vagrant reload --provision` command. This will ensure that your new aliases are available on the machine.

Updating Homestead

Before you begin updating Homestead you should ensure you have removed your current virtual machine by running the following command in your Homestead directory:

```
vagrant destroy
```

Next, you need to update the Homestead source code. If you cloned the repository, you can execute the following commands at the location you originally cloned the repository:

```
git fetch
git pull origin release
```

These commands pull the latest Homestead code from the GitHub repository, fetch the latest tags, and then check out the latest tagged release. You can find the latest stable release version on Homestead's [GitHub releases page](#).

If you have installed Homestead via your project's `composer.json` file, you should ensure your `composer.json` file contains `"laravel/homestead": "^12"` and update your dependencies:

```
composer update
```

Next, you should update the Vagrant box using the `vagrant box update` command:

```
vagrant box update
```

After updating the Vagrant box, you should run the `bash init.sh` command from the Homestead directory in order to update Homestead's additional configuration files. You will be asked whether you wish to overwrite your existing `Homestead.yaml`, `after.sh`, and `aliases` files:

```
# macOS / Linux...
bash init.sh

# Windows...
init.bat
```

Finally, you will need to regenerate your Homestead virtual machine to utilize the latest Vagrant installation:

```
vagrant up
```

Daily Usage

Connecting via SSH

You can SSH into your virtual machine by executing the `vagrant ssh` terminal command from your Homestead directory.

Adding Additional Sites

Once your Homestead environment is provisioned and running, you may want to add additional Nginx sites for your other Laravel projects. You can run as many Laravel projects as you wish on a single Homestead environment. To add an additional site, add the site to your `Homestead.yaml` file.

```
sites:
  - map: homestead.test
    to: /home/vagrant/project1/public
  - map: another.test
    to: /home/vagrant/project2/public
```



You should ensure that you have configured a [folder mapping](#) for the project's directory before adding the site.

If Vagrant is not automatically managing your "hosts" file, you may need to add the new site to that file as well. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at

`C:\Windows\System32\drivers\etc\hosts`:

```
192.168.56.56 homestead.test
192.168.56.56 another.test
```

Once the site has been added, execute the `vagrant reload --provision` terminal command from your Homestead directory.

Site Types

Homestead supports several "types" of sites which allow you to easily run projects that are not based on Laravel. For example, we may easily add a Statamic application to Homestead using the `statamic` site type:

```
sites:
  - map: statamic.test
    to: /home/vagrant/my-symfony-project/web
    type: "statamic"
```

The available site types are: `apache`, `apache-proxy`, `apigility`, `expressive`, `laravel` (the default), `proxy` (for nginx), `silverstripe`, `statamic`, `symfony2`, `symfony4`, and `zf`.

Site Parameters

You may add additional Nginx `fastcgi_param` values to your site via the `params` site directive:

```
sites:
  - map: homestead.test
    to: /home/vagrant/project1/public
    params:
      - key: FOO
        value: BAR
```

Environment Variables

You can define global environment variables by adding them to your `Homestead.yaml` file:

```
variables:
  - key: APP_ENV
    value: local
  - key: FOO
    value: bar
```

After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing the `vagrant reload --provision` command. This will update the PHP-FPM configuration for all of the installed PHP versions and also update the environment for the `vagrant` user.

Ports

By default, the following ports are forwarded to your Homestead environment:

- **HTTP:** 8000 → Forwards To 80
- **HTTPS:** 44300 → Forwards To 443

Forwarding Additional Ports

If you wish, you may forward additional ports to the Vagrant box by defining a `ports` configuration entry within your `Homestead.yaml` file. After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing the `vagrant reload --provision` command:

```
ports:
  - send: 50000
    to: 5000
  - send: 7777
    to: 777
    protocol: udp
```

Below is a list of additional Homestead service ports that you may wish to map from your host machine to your Vagrant box:

- **SSH:** 2222 → To 22
- **ngrok UI:** 4040 → To 4040
- **MySQL:** 33060 → To 3306
- **PostgreSQL:** 54320 → To 5432
- **MongoDB:** 27017 → To 27017
- **Mailpit:** 8025 → To 8025

- **Minio:** 9600 → To 9600

PHP Versions

Homestead supports running multiple versions of PHP on the same virtual machine. You may specify which version of PHP to use for a given site within your `Homestead.yaml` file. The available PHP versions are: "5.6", "7.0", "7.1", "7.2", "7.3", "7.4", "8.0", "8.1", "8.2", and "8.3", (the default):

```
sites:
  - map: homestead.test
    to: /home/vagrant/project1/public
    php: "7.1"
```

Within your Homestead virtual machine, you may use any of the supported PHP versions via the CLI:

```
php5.6 artisan list
php7.0 artisan list
php7.1 artisan list
php7.2 artisan list
php7.3 artisan list
php7.4 artisan list
php8.0 artisan list
php8.1 artisan list
php8.2 artisan list
php8.3 artisan list
```

You may change the default version of PHP used by the CLI by issuing the following commands from within your Homestead virtual machine:

```
php56
php70
php71
php72
php73
php74
php80
php81
php82
php83
```

Connecting to Databases

A `homestead` database is configured for both MySQL and PostgreSQL out of the box. To connect to your MySQL or PostgreSQL database from your host machine's database client, you should connect to `127.0.0.1` on port `33060` (MySQL) or `54320` (PostgreSQL). The username and password for both databases is `homestead` / `secret`.



You should only use these non-standard ports when connecting to the databases from your host machine. You will use the default 3306 and 5432 ports in your Laravel application's `database` configuration file since Laravel is running within the virtual machine.

Database Backups

Homestead can automatically backup your database when your Homestead virtual machine is destroyed. To utilize this feature, you must be using Vagrant 2.1.0 or greater. Or, if you are using an older version of Vagrant, you must install the `vagrant-triggers` plug-in. To enable automatic database backups, add the following line to your `Homestead.yaml` file:

```
backup: true
```

Once configured, Homestead will export your databases to `.backup/mysql_backup` and `.backup/postgres_backup` directories when the `vagrant destroy` command is executed. These directories can be found in the folder where you installed Homestead or in the root of your project if you are using the per project installation method.

Configuring Cron Schedules

Laravel provides a convenient way to schedule cron jobs by scheduling a single `schedule:run` Artisan command to run every minute. The `schedule:run` command will examine the job schedule defined in your `[App\Console\Kernel]` class to determine which scheduled tasks to run.

If you would like the `schedule:run` command to be run for a Homestead site, you may set the `schedule` option to `true` when defining the site:

```
sites:
- map: homestead.test
  to: /home/vagrant/project1/public
  schedule: true
```

The cron job for the site will be defined in the `/etc/cron.d` directory of the Homestead virtual machine.

Configuring Mailpit

Mailpit allows you to intercept your outgoing email and examine it without actually sending the mail to its recipients. To get started, update your application's `.env` file to use the following mail settings:

```
MAIL_MAILER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Once Mailpit has been configured, you may access the Mailpit dashboard at `http://localhost:8025`.

Configuring Minio

Minio is an open source object storage server with an Amazon S3 compatible API. To install Minio, update your `Homestead.yaml` file with the following configuration option in the features section:

```
minio: true
```

By default, Minio is available on port 9600. You may access the Minio control panel by visiting `http://localhost:9600`. The default access key is `homestead`, while the default secret key is `secretkey`. When accessing Minio, you should always use region `us-east-1`.

In order to use Minio, you will need to adjust the S3 disk configuration in your application's `config/filesystems.php` configuration file. You will need to add the `use_path_style_endpoint` option to the disk configuration as well as change the `url` key to `endpoint`:

```
's3' => [
    'driver' => 's3',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION'),
    'bucket' => env('AWS_BUCKET'),
    'endpoint' => env('AWS_URL'),
    'use_path_style_endpoint' => true,
]
```

Finally, ensure your `.env` file has the following options:

```
AWS_ACCESS_KEY_ID=homestead
AWS_SECRET_ACCESS_KEY=secretkey
AWS_DEFAULT_REGION=us-east-1
AWS_URL=http://localhost:9600
```

To provision Minio powered "S3" buckets, add a `buckets` directive to your `Homestead.yaml` file. After defining your buckets, you should execute the `vagrant reload --provision` command in your terminal:

```
buckets:
  - name: your-bucket
    policy: public
  - name: your-private-bucket
    policy: none
```

Supported `policy` values include: `none`, `download`, `upload`, and `public`.

Laravel Dusk

In order to run [Laravel Dusk](#) tests within Homestead, you should enable the [webdriver feature](#) in your Homestead configuration:

```
features:
  - webdriver: true
```

After enabling the `webdriver` feature, you should execute the `vagrant reload --provision` command in your terminal.

Sharing Your Environment

Sometimes you may wish to share what you're currently working on with coworkers or a client. Vagrant has built-in support for this via the `vagrant share` command; however, this will not work if you have multiple sites configured in your `Homestead.yaml` file.

To solve this problem, Homestead includes its own `share` command. To get started, [SSH into your Homestead virtual machine](#) via `vagrant ssh` and execute the `share homestead.test` command. This command will share the `homestead.test` site from your `Homestead.yaml` configuration file. You may substitute any of your other configured sites for `homestead.test`:

```
share homestead.test
```

After running the command, you will see an Ngrok screen appear which contains the activity log and the publicly accessible URLs for the shared site. If you would like to specify a custom region, subdomain, or other Ngrok runtime

option, you may add them to your `share` command:

```
share homestead.test --region=eu --subdomain=laravel
```

If you need to share content over HTTPS rather than HTTP, using the `sshare` command instead of `share` will enable you to do so.



Remember, Vagrant is inherently insecure and you are exposing your virtual machine to the Internet when running the `share` command.

Debugging and Profiling

Debugging Web Requests With Xdebug

Homestead includes support for step debugging using [Xdebug](#). For example, you can access a page in your browser and PHP will connect to your IDE to allow inspection and modification of the running code.

By default, Xdebug is already running and ready to accept connections. If you need to enable Xdebug on the CLI, execute the `sudo phpenmod xdebug` command within your Homestead virtual machine. Next, follow your IDE's instructions to enable debugging. Finally, configure your browser to trigger Xdebug with an extension or [bookmarklet](#).



Xdebug causes PHP to run significantly slower. To disable Xdebug, run `sudo phpdismod xdebug` within your Homestead virtual machine and restart the FPM service.

Autostarting Xdebug

When debugging functional tests that make requests to the web server, it is easier to autostart debugging rather than modifying tests to pass through a custom header or cookie to trigger debugging. To force Xdebug to start automatically, modify the `/etc/php/7.x/fpm/conf.d/20-xdebug.ini` file inside your Homestead virtual machine and add the following configuration:

```
; If Homestead.yaml contains a different subnet for the IP address, this address may be
; different...
xdebug.client_host = 192.168.10.1
xdebug.mode = debug
xdebug.start_with_request = yes
```

Debugging CLI Applications

To debug a PHP CLI application, use the `xphp` shell alias inside your Homestead virtual machine:

```
xphp /path/to/script
```

Profiling Applications With Blackfire

[Blackfire](#) is a service for profiling web requests and CLI applications. It offers an interactive user interface which displays profile data in call-graphs and timelines. It is built for use in development, staging, and production, with no overhead for end users. In addition, Blackfire provides performance, quality, and security checks on code and `php.ini` configuration settings.

The [Blackfire Player](#) is an open-source Web Crawling, Web Testing, and Web Scraping application which can work jointly with Blackfire in order to script profiling scenarios.

To enable Blackfire, use the "features" setting in your Homestead configuration file:

```
features:
  - blackfire:
      server_id: "server_id"
      server_token: "server_value"
      client_id: "client_id"
      client_token: "client_value"
```

Blackfire server credentials and client credentials [require a Blackfire account](#). Blackfire offers various options to profile an application, including a CLI tool and browser extension. Please [review the Blackfire documentation for more details](#).

Network Interfaces

The `networks` property of the `Homestead.yaml` file configures network interfaces for your Homestead virtual machine. You may configure as many interfaces as necessary:

```
networks:
  - type: "private_network"
    ip: "192.168.10.20"
```

To enable a [bridged](#) interface, configure a `bridge` setting for the network and change the network type to `public_network`:

```
networks:
  - type: "public_network"
    ip: "192.168.10.20"
    bridge: "en1: Wi-Fi (AirPort)"
```

To enable [DHCP](#), just remove the `ip` option from your configuration:

```
networks:
  - type: "public_network"
    bridge: "en1: Wi-Fi (AirPort)"
```

To update what device the network is using, you may add a `dev` option to the network's configuration. The default `dev` value is `eth0`:

```
networks:
  - type: "public_network"
    ip: "192.168.10.20"
    bridge: "en1: Wi-Fi (AirPort)"
    dev: "enp2s0"
```

Extending Homestead

You may extend Homestead using the `after.sh` script in the root of your Homestead directory. Within this file, you may add any shell commands that are necessary to properly configure and customize your virtual machine.

When customizing Homestead, Ubuntu may ask you if you would like to keep a package's original configuration or overwrite it with a new configuration file. To avoid this, you should use the following command when installing packages in order to avoid overwriting any configuration previously written by Homestead:

```
sudo apt-get -y \
-o Dpkg::Options::="--force-confdef" \
-o Dpkg::Options::="--force-confold" \
install package-name
```

User Customizations

When using Homestead with your team, you may want to tweak Homestead to better fit your personal development style. To accomplish this, you may create a `user-customizations.sh` file in the root of your Homestead directory (the same directory containing your `Homestead.yaml` file). Within this file, you may make any customization you would like; however, the `user-customizations.sh` should not be version controlled.

Provider Specific Settings

VirtualBox

`natdnshostresolver`

By default, Homestead configures the `natdnshostresolver` setting to `on`. This allows Homestead to use your host operating system's DNS settings. If you would like to override this behavior, add the following configuration options to your `Homestead.yaml` file:

```
provider: virtualbox
natdnshostresolver: 'off'
```

Laravel Horizon

- [Introduction](#)
- [Installation](#)
 - [Configuration](#)
 - [Balancing Strategies](#)
 - [Dashboard Authorization](#)
 - [Silenced Jobs](#)
- [Upgrading Horizon](#)
- [Running Horizon](#)
 - [Deploying Horizon](#)
- [Tags](#)
- [Notifications](#)
- [Metrics](#)
- [Deleting Failed Jobs](#)
- [Clearing Jobs From Queues](#)

Introduction



Before digging into Laravel Horizon, you should familiarize yourself with Laravel's base [queue services](#). Horizon augments Laravel's queue with additional features that may be confusing if you are not already familiar with the basic queue features offered by Laravel.

[Laravel Horizon](#) provides a beautiful dashboard and code-driven configuration for your Laravel powered [Redis queues](#). Horizon allows you to easily monitor key metrics of your queue system such as job throughput, runtime, and job failures.

When using Horizon, all of your queue worker configuration is stored in a single, simple configuration file. By defining your application's worker configuration in a version controlled file, you may easily scale or modify your application's queue workers when deploying your application.

The screenshot shows the Laravel Horizon dashboard with the following data:

Jobs Per Minute	Jobs Past Hour	Failed Jobs Past 7 Days	Status
246	1,100	0	Active

Total Processes	Max Wait Time	Max Runtime	Max Throughput
2	-	default	default

Current Workload

Queue	Jobs	Processes	Wait
default	200	2	A few seconds

nunomaduro/home-xaZQ

Supervisor	Queues	Processes	Balancing
supervisor-1	default	2	Auto

Installation



Laravel Horizon requires that you use [Redis](#) to power your queue. Therefore, you should ensure that your queue connection is set to `redis` in your application's `config/queue.php` configuration file.

You may install Horizon into your project using the Composer package manager:

```
composer require laravel/horizon
```

After installing Horizon, publish its assets using the `horizon:install` Artisan command:

```
php artisan horizon:install
```

Configuration

After publishing Horizon's assets, its primary configuration file will be located at `config/horizon.php`. This configuration file allows you to configure the queue worker options for your application. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.



Horizon uses a Redis connection named `horizon` internally. This Redis connection name is reserved and should not be assigned to another Redis connection in the `database.php` configuration file or as the value of the `use` option in the `horizon.php` configuration file.

Environments

After installation, the primary Horizon configuration option that you should familiarize yourself with is the `environments` configuration option. This configuration option is an array of environments that your application runs on and defines the worker process options for each environment. By default, this entry contains a `production` and `local` environment. However, you are free to add more environments as needed:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
        ],
    ],
    'local' => [
        'supervisor-1' => [
            'maxProcesses' => 3,
        ],
    ],
],
```

When you start Horizon, it will use the worker process configuration options for the environment that your application is running on. Typically, the environment is determined by the value of the `APP_ENV` [environment variable](#). For example, the default `local` Horizon environment is configured to start three worker processes and automatically balance the number of worker processes assigned to each queue. The default `production` environment is configured to start a maximum of 10 worker processes and automatically balance the number of worker processes assigned to each queue.



You should ensure that the `environments` portion of your `horizon` configuration file contains an entry for each [environment](#) on which you plan to run Horizon.

Supervisors

As you can see in Horizon's default configuration file, each environment can contain one or more "supervisors". By default, the configuration file defines this supervisor as `supervisor-1`; however, you are free to name your supervisors whatever you want. Each supervisor is essentially responsible for "supervising" a group of worker processes and takes care of balancing worker processes across queues.

You may add additional supervisors to a given environment if you would like to define a new group of worker processes that should run in that environment. You may choose to do this if you would like to define a different balancing strategy or worker process count for a given queue used by your application.

Maintenance Mode

While your application is in [maintenance mode](#), queued jobs will not be processed by Horizon unless the supervisor's `force` option is defined as `true` within the Horizon configuration file:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            // ...
            'force' => true,
        ],
    ],
],
```

Default Values

Within Horizon's default configuration file, you will notice a `defaults` configuration option. This configuration option specifies the default values for your application's [supervisors](#). The supervisor's default configuration values will be merged into the supervisor's configuration for each environment, allowing you to avoid unnecessary repetition when defining your supervisors.

Balancing Strategies

Unlike Laravel's default queue system, Horizon allows you to choose from three worker balancing strategies: `simple`, `auto`, and `false`. The `simple` strategy splits incoming jobs evenly between worker processes:

```
'balance' => 'simple',
```

The `auto` strategy, which is the configuration file's default, adjusts the number of worker processes per queue based on the current workload of the queue. For example, if your `notifications` queue has 1,000 pending jobs while your `render` queue is empty, Horizon will allocate more workers to your `notifications` queue until the queue is empty.

When using the `auto` strategy, you may define the `minProcesses` and `maxProcesses` configuration options to control the minimum and the maximum number of worker processes Horizon should scale up and down to:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default'],
            'balance' => 'auto',
            'autoScalingStrategy' => 'time',
            'minProcesses' => 1,
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
            'tries' => 3,
        ],
    ],
],
```

The `autoScalingStrategy` configuration value determines if Horizon will assign more worker processes to queues based on the total amount of time it will take to clear the queue (`time` strategy) or by the total number of jobs on the queue (`size` strategy).

The `balanceMaxShift` and `balanceCooldown` configuration values determine how quickly Horizon will scale to meet worker demand. In the example above, a maximum of one new process will be created or destroyed every three seconds.

You are free to tweak these values as necessary based on your application's needs.

When the `balance` option is set to `false`, the default Laravel behavior will be used, wherein queues are processed in the order they are listed in your configuration.

Dashboard Authorization

The Horizon dashboard may be accessed via the `/horizon` route. By default, you will only be able to access this dashboard in the `local` environment. However, within your `app/Providers/HorizonServiceProvider.php` file, there is an [authorization gate](#) definition. This authorization gate controls access to Horizon in **non-local** environments. You are free to modify this gate as needed to restrict access to your Horizon installation:

```
/**
 * Register the Horizon gate.
 *
 * This gate determines who can access Horizon in non-local environments.
 */
protected function gate(): void
{
    Gate::define('viewHorizon', function (User $user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

Alternative Authentication Strategies

Remember that Laravel automatically injects the authenticated user into the gate closure. If your application is providing Horizon security via another method, such as IP restrictions, then your Horizon users may not need to "login". Therefore, you will need to change `function (User $user)` closure signature above to `function (User $user = null)` in order to force Laravel to not require authentication.

Silenced Jobs

Sometimes, you may not be interested in viewing certain jobs dispatched by your application or third-party packages. Instead of these jobs taking up space in your "Completed Jobs" list, you can silence them. To get started, add the job's class name to the `silenced` configuration option in your application's `horizon` configuration file:

```
'silenced' => [
    App\Jobs\ProcessPodcast::class,
],
```

Alternatively, the job you wish to silence can implement the `Laravel\Horizon\Contracts\Silenced` interface. If a job implements this interface, it will automatically be silenced, even if it is not present in the `silenced` configuration array:

```
use Laravel\Horizon\Contracts\Silenced;

class ProcessPodcast implements ShouldQueue, Silenced
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    // ...
}
```

Upgrading Horizon

When upgrading to a new major version of Horizon, it's important that you carefully review [the upgrade guide](#). In addition, when upgrading to any new Horizon version, you should re-publish Horizon's assets:

```
php artisan horizon:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `vendor:publish --tag=laravel-assets` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan vendor:publish --tag=laravel-assets --ansi --force"
        ]
    }
}
```

Running Horizon

Once you have configured your supervisors and workers in your application's `config/horizon.php` configuration file, you may start Horizon using the `horizon` Artisan command. This single command will start all of the configured worker processes for the current environment:

```
php artisan horizon
```

You may pause the Horizon process and instruct it to continue processing jobs using the `horizon:pause` and `horizon:continue` Artisan commands:

```
php artisan horizon:pause

php artisan horizon:continue
```

You may also pause and continue specific Horizon [supervisors](#) using the `horizon:pause-supervisor` and `horizon:continue-supervisor` Artisan commands:

```
php artisan horizon:pause-supervisor supervisor-1

php artisan horizon:continue-supervisor supervisor-1
```

You may check the current status of the Horizon process using the `horizon:status` Artisan command:

```
php artisan horizon:status
```

You may gracefully terminate the Horizon process using the `horizon:terminate` Artisan command. Any jobs that are currently being processed by will be completed and then Horizon will stop executing:

```
php artisan horizon:terminate
```

Deploying Horizon

When you're ready to deploy Horizon to your application's actual server, you should configure a process monitor to monitor the `php artisan horizon` command and restart it if it exits unexpectedly. Don't worry, we'll discuss how to install a process monitor below.

During your application's deployment process, you should instruct the Horizon process to terminate so that it will be restarted by your process monitor and receive your code changes:

```
php artisan horizon:terminate
```

Installing Supervisor

Supervisor is a process monitor for the Linux operating system and will automatically restart your `horizon` process if it stops executing. To install Supervisor on Ubuntu, you may use the following command. If you are not using Ubuntu, you can likely install Supervisor using your operating system's package manager:

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



If configuring Supervisor yourself sounds overwhelming, consider using [Laravel Forge](#), which will automatically install and configure Supervisor for your Laravel projects.

Supervisor Configuration

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

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/example.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/example.com/horizon.log
stopwaitsecs=3600
```

When defining your Supervisor configuration, you should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.



While the examples above are valid for Ubuntu based servers, the location and file extension expected of Supervisor configuration files may vary between other server operating systems. Please consult your server's documentation for more information.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the monitored processes using the following commands:

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start horizon
```



For more information on running Supervisor, consult the [Supervisor documentation](#).

Tags

Horizon allows you to assign “tags” to jobs, including mailables, broadcast events, notifications, and queued event listeners. In fact, Horizon will intelligently and automatically tag most jobs depending on the Eloquent models that are attached to the job. For example, take a look at the following job:

```
<?php

namespace App\Jobs;

use App\Models\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct(
        public Video $video,
    ) {}

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

If this job is queued with an `App\Models\Video` instance that has an `id` attribute of `1`, it will automatically receive the tag `App\Models\Video:1`. This is because Horizon will search the job's properties for any Eloquent models. If Eloquent models are found, Horizon will intelligently tag the job using the model's class name and primary key:

```
use App\Jobs\RenderVideo;
use App\Models\Video;

$video = Video::find(1);

RenderVideo::dispatch($video);
```

Manually Tagging Jobs

If you would like to manually define the tags for one of your queueable objects, you may define a `tags` method on the class:

```
class RenderVideo implements ShouldQueue
{
    /**
     * Get the tags that should be assigned to the job.
     *
     * @return array<int, string>
     */
    public function tags(): array
    {
        return ['render', 'video:' . $this->video->id];
    }
}
```

Manually Tagging Event Listeners

When retrieving the tags for a queued event listener, Horizon will automatically pass the event instance to the `tags` method, allowing you to add event data to the tags:

```
class SendRenderNotifications implements ShouldQueue
{
    /**
     * Get the tags that should be assigned to the listener.
     *
     * @return array<int, string>
     */
    public function tags(VideoRendered $event): array
    {
        return ['video:' . $event->video->id];
    }
}
```

Notifications



When configuring Horizon to send Slack or SMS notifications, you should review the [prerequisites for the relevant notification channel](#).

If you would like to be notified when one of your queues has a long wait time, you may use the `Horizon::routeMailNotificationsTo`, `Horizon::routeSlackNotificationsTo`, and

`Horizon::routeSmsNotificationsTo` methods. You may call these methods from the `boot` method of your application's `App\Providers\HorizonServiceProvider`:

```
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    parent::boot();

    Horizon::routeSmsNotificationsTo('15556667777');
    Horizon::routeMailNotificationsTo('example@example.com');
    Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
}
```

Configuring Notification Wait Time Thresholds

You may configure how many seconds are considered a "long wait" within your application's `config/horizon.php` configuration file. The `waits` configuration option within this file allows you to control the long wait threshold for each connection / queue combination. Any undefined connection / queue combinations will default to a long wait threshold of 60 seconds:

```
'waits' => [
    'redis:critical' => 30,
    'redis:default' => 60,
    'redis:batch' => 120,
],
```

Metrics

Horizon includes a metrics dashboard which provides information regarding your job and queue wait times and throughput. In order to populate this dashboard, you should configure Horizon's `snapshot` Artisan command to run every five minutes via your application's `scheduler`:

```
/**
 * Define the application's command schedule.
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

Deleting Failed Jobs

If you would like to delete a failed job, you may use the `horizon:forget` command. The `horizon:forget` command accepts the ID or UUID of the failed job as its only argument:

```
php artisan horizon:forget 5
```

Clearing Jobs From Queues

If you would like to delete all jobs from your application's default queue, you may do so using the `horizon:clear` Artisan command:

```
php artisan horizon:clear
```

You may provide the `queue` option to delete jobs from a specific queue:

```
php artisan horizon:clear --queue=emails
```

Laravel MCP

- [Introduction](#)
- [Installation](#)
 - [Publishing Routes](#)
- [Creating Servers](#)
 - [Server Registration](#)
 - [Web Servers](#)
 - [Local Servers](#)
- [Tools](#)
 - [Creating Tools](#)
 - [Tool Input Schemas](#)
 - [Validating Tool Arguments](#)
 - [Tool Dependency Injection](#)
 - [Tool Annotations](#)
 - [Conditional Tool Registration](#)
 - [Tool Responses](#)
- [Prompts](#)
 - [Creating Prompts](#)
 - [Prompt Arguments](#)
 - [Validating Prompt Arguments](#)
 - [Prompt Dependency Injection](#)
 - [Conditional Prompt Registration](#)
 - [Prompt Responses](#)
- [Resources](#)
 - [Creating Resources](#)
 - [Resource URI and MIME Type](#)
 - [Resource Request](#)
 - [Resource Dependency Injection](#)
 - [Conditional Resource Registration](#)
 - [Resource Responses](#)
- [Authentication](#)
 - [OAuth 2.1](#)
 - [Sanctum](#)
- [Authorization](#)
- [Testing Servers](#)
 - [MCP Inspector](#)
 - [Unit Tests](#)

Introduction

[Laravel MCP](#) provides a simple and elegant way for AI clients to interact with your Laravel application through the [Model Context Protocol](#). It offers an expressive, fluent interface for defining servers, tools, resources, and prompts that enable AI-powered interactions with your application.

Installation

To get started, install Laravel MCP into your project using the Composer package manager:

```
composer require laravel/mcp
```

Publishing Routes

After installing Laravel MCP, execute the `vendor:publish` Artisan command to publish the `routes/ai.php` file where you will define your MCP servers:

```
php artisan vendor:publish --tag=ai-routes
```

This command creates the `routes/ai.php` file in your application's `routes` directory, which you will use to register your MCP servers.

Creating Servers

You can create an MCP server using the `make:mcp-server` Artisan command. Servers act as the central communication point that exposes MCP capabilities like tools, resources, and prompts to AI clients:

```
php artisan make:mcp-server WeatherServer
```

This command will create a new server class in the `app/Mcp/Servers` directory. The generated server class extends Laravel MCP's base `Laravel\Mcp\Server` class and provides properties for registering tools, resources, and prompts:

```
<?php

namespace App\Mcp\Servers;

use Laravel\Mcp\Server;

class WeatherServer extends Server
{
    /**
     * The tools registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Tool>>
     */
    protected array $tools = [
        // ExampleTool::class,
    ];

    /**
     * The resources registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Resource>>
     */
    protected array $resources = [
        // ExampleResource::class,
    ];

    /**
     * The prompts registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Prompt>>
     */
    protected array $prompts = [
        // ExamplePrompt::class,
    ];
}
```

Server Registration

Once you've created a server, you must register it in your `routes/ai.php` file to make it accessible. Laravel MCP provides two methods for registering servers: `web` for HTTP-accessible servers and `local` for command-line servers.

Web Servers

Web servers are the most common types of servers and are accessible via HTTP POST requests, making them ideal for remote AI clients or web-based integrations. Register a web server using the `web` method:

```
use App\Mcp\Servers\WeatherServer;
use Laravel\Mcp\Facades\Mcp;

Mcp::web('/mcp/weather', WeatherServer::class);
```

Just like normal routes, you may apply middleware to protect your web servers:

```
Mcp::web('/mcp/weather', WeatherServer::class)
->middleware(['throttle:mcp']);
```

Local Servers

Local servers run as Artisan commands, perfect for development, testing, or local AI assistant integrations. Register a local server using the `local` method:

```
use App\Mcp\Servers\WeatherServer;
use Laravel\Mcp\Facades\Mcp;

Mcp::local('weather', WeatherServer::class);
```

Once registered, you should not typically need to manually run `mcp:start` yourself. Instead, configure your MCP client (AI agent) to start the server. The `mcp:start` command is designed to be invoked by the client, which will handle starting and stopping the server as needed:

```
php artisan mcp:start weather
```

Tools

Tools enable your server to expose functionality that AI clients can call. They allow language models to perform actions, run code, or interact with external systems.

Creating Tools

To create a tool, run the `make:mcp-tool` Artisan command:

```
php artisan make:mcp-tool CurrentWeatherTool
```

After creating a tool, register it in your server's `$tools` property:

```
<?php

namespace App\Mcp\Servers;

use App\Mcp\Tools\CurrentWeatherTool;
use Laravel\Mcp\Server;

class WeatherServer extends Server
{
    /**
     * The tools registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Tool>>
     */
    protected array $tools = [
        CurrentWeatherTool::class,
    ];
}
```

Tool Name, Title, and Description

By default, the tool's name and title are derived from the class name. For example, `CurrentWeatherTool` will have a name of `current-weather` and a title of `Current Weather Tool`. You may customize these values by defining the tool's `$name` and `$title` properties:

```
class CurrentWeatherTool extends Tool
{
    /**
     * The tool's name.
     */
    protected string $name = 'get-optimistic-weather';

    /**
     * The tool's title.
     */
    protected string $title = 'Get Optimistic Weather Forecast';

    // ...
}
```

Tool descriptions are not automatically generated. You should always provide a meaningful description by defining a `$description` property on your tool:

```
class CurrentWeatherTool extends Tool
{
    /**
     * The tool's description.
     */
    protected string $description = 'Fetches the current weather forecast for a specified
location.';

    //
}
```



The description is a critical part of the tool's metadata, as it helps AI models understand when and how to use the tool effectively.

Tool Input Schemas

Tools can define input schemas to specify what arguments they accept from AI clients. Use Laravel's `Illuminate\JsonSchema\JsonSchema` builder to define your tool's input requirements:

```
<?php

namespace App\Mcp\Tools;

use Illuminate\JsonSchema\JsonSchema;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Get the tool's input schema.
     *
     * @return array<string, JsonSchema>
     */
    public function schema(JsonSchema $schema): array
    {
        return [
            'location' => $schema->string()
                ->description('The location to get the weather for.')
                ->required(),

            'units' => $schema->enum(['celsius', 'fahrenheit'])
                ->description('The temperature units to use.')
                ->default('celsius'),
        ];
    }
}
```

Validating Tool Arguments

JSON Schema definitions provide a basic structure for tool arguments, but you may also want to enforce more complex validation rules.

Laravel MCP integrates seamlessly with Laravel's [validation features](#). You may validate incoming tool arguments within your tool's `handle` method:

```
<?php

namespace App\Mcp\Tools;

use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Handle the tool request.
     */
    public function handle(Request $request): Response
    {
        $validated = $request->validate([
            'location' => 'required|string|max:100',
            'units' => 'in:celsius,fahrenheit',
        ]);

        // Fetch weather data using the validated arguments...
    }
}
```

On validation failure, AI clients will act based on the error messages you provide. As such, it is critical to provide clear and actionable error messages:

```
$validated = $request->validate([
    'location' => ['required','string','max:100'],
    'units' => 'in:celsius,fahrenheit',
], [
    'location.required' => 'You must specify a location to get the weather for. For example, "New York City" or "Tokyo".',
    'units.in' => 'You must specify either "celsius" or "fahrenheit" for the units.',
]);
```

Tool Dependency Injection

The Laravel [service container](#) is used to resolve all tools. As a result, you are able to type-hint any dependencies your tool may need in its constructor. The declared dependencies will automatically be resolved and injected into the tool instance:

```
<?php

namespace App\Mcp\Tools;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Create a new tool instance.
     */
    public function __construct(
        protected WeatherRepository $weather,
    ) {}

    // ...
}
```

In addition to constructor injection, you may also type-hint dependencies in your tool's `handle()` method. The service container will automatically resolve and inject the dependencies when the method is called:

```
<?php

namespace App\Mcp\Tools;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Handle the tool request.
     */
    public function handle(Request $request, WeatherRepository $weather): Response
    {
        $location = $request->get('location');

        $forecast = $weather->getForecastFor($location);

        // ...
    }
}
```

Tool Annotations

You may enhance your tools with [annotations](#) to provide additional metadata to AI clients. These annotations help AI models understand the tool's behavior and capabilities. Annotations are added to tools via attributes:

```
<?php

namespace App\Mcp\Tools;

use Laravel\Mcp\Server\Tools\Annotations\IsIdempotent;
use Laravel\Mcp\Server\Tools\Annotations\IsReadOnly;
use Laravel\Mcp\Server\Tool;

#[IsIdempotent]
#[IsReadOnly]
class CurrentWeatherTool extends Tool
{
    //
}
```

Available annotations include:

Annotation	Type	Description
#[IsReadOnly]	boolean	Indicates the tool does not modify its environment.
# [IsDestructive]	boolean	Indicates the tool may perform destructive updates (only meaningful when not read-only).
#[IsIdempotent]	boolean	Indicates repeated calls with same arguments have no additional effect (when not read-only).
#[IsOpenWorld]	boolean	Indicates the tool may interact with external entities.

Conditional Tool Registration

You may conditionally register tools at runtime by implementing the `shouldRegister` method in your tool class. This method allows you to determine whether a tool should be available based on application state, configuration, or request parameters:

```
<?php

namespace App\Mcp\Tools;

use Laravel\Mcp\Request;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Determine if the tool should be registered.
     */
    public function shouldRegister(Request $request): bool
    {
        return $request?->user()?->subscribed() ?? false;
    }
}
```

When a tool's `shouldRegister` method returns `false`, it will not appear in the list of available tools and cannot be invoked by AI clients.

Tool Responses

Tools must return an instance of `Laravel\Mcp\Response`. The Response class provides several convenient methods for creating different types of responses:

For simple text responses, use the `text` method:

```
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;

/**
 * Handle the tool request.
 */
public function handle(Request $request): Response
{
    // ...

    return Response::text('Weather Summary: Sunny, 72°F');
}
```

To indicate an error occurred during tool execution, use the `error` method:

```
return Response::error('Unable to fetch weather data. Please try again.');
```

Multiple Content Responses

Tools can return multiple pieces of content by returning an array of `Response` instances:

```
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;

/**
 * Handle the tool request.
 *
 * @return array<int, \Laravel\Mcp\Response>
 */
public function handle(Request $request): array
{
    // ...

    return [
        Response::text('Weather Summary: Sunny, 72°F'),
        Response::text('***Detailed Forecast***\n- Morning: 65°F\n- Afternoon: 78°F\n- Evening: 70°F')
    ];
}
```

Streaming Responses

For long-running operations or real-time data streaming, tools can return a [generator](#) from their `handle` method. This enables sending intermediate updates to the client before the final response:

```
<?php

namespace App\Mcp\Tools;

use Generator;
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Tool;

class CurrentWeatherTool extends Tool
{
    /**
     * Handle the tool request.
     *
     * @return \Generator<int, \Laravel\Mcp\Response>
     */
    public function handle(Request $request): Generator
    {
        $locations = $request->array('locations');

        foreach ($locations as $index => $location) {
            yield Response::notification('processing/progress', [
                'current' => $index + 1,
                'total' => count($locations),
                'location' => $location,
            ]);

            yield Response::text($this->forecastFor($location));
        }
    }
}
```

When using web-based servers, streaming responses automatically open an SSE (Server-Sent Events) stream, sending each yielded message as an event to the client.

Prompts

Prompts enable your server to share reusable prompt templates that AI clients can use to interact with language models. They provide a standardized way to structure common queries and interactions.

Creating Prompts

To create a prompt, run the `make:mcp-prompt` Artisan command:

```
php artisan make:mcp-prompt DescribeWeatherPrompt
```

After creating a prompt, register it in your server's `$prompts` property:

```
<?php

namespace App\Mcp\Servers;

use App\Mcp\Prompts\DescribeWeatherPrompt;
use Laravel\Mcp\Server;

class WeatherServer extends Server
{
    /**
     * The prompts registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Prompt>>
     */
    protected array $prompts = [
        DescribeWeatherPrompt::class,
    ];
}
```

Prompt Name, Title, and Description

By default, the prompt's name and title are derived from the class name. For example, `DescribeWeatherPrompt` will have a name of `describe-weather` and a title of `Describe Weather Prompt`. You may customize these values by defining `$name` and `$title` properties on your prompt:

```
class DescribeWeatherPrompt extends Prompt
{
    /**
     * The prompt's name.
     */
    protected string $name = 'weather-assistant';

    /**
     * The prompt's title.
     */
    protected string $title = 'Weather Assistant Prompt';

    // ...
}
```

Prompt descriptions are not automatically generated. You should always provide a meaningful description by defining a `$description` property on your prompts:

```
class DescribeWeatherPrompt extends Prompt
{
    /**
     * The prompt's description.
     */
    protected string $description = 'Generates a natural-language explanation of the weather for a given location.';

    //
}
```



The description is a critical part of the prompt's metadata, as it helps AI models understand when and how to get the best use out of the prompt.

Prompt Arguments

Prompts can define arguments that allow AI clients to customize the prompt template with specific values. Use the `arguments` method to define what arguments your prompt accepts:

```
<?php

namespace App\Mcp\Prompts;

use Laravel\Mcp\Server\Prompt;
use Laravel\Mcp\Server\Prompts\Argument;

class DescribeWeatherPrompt extends Prompt
{
    /**
     * Get the prompt's arguments.
     *
     * @return array<int, \Laravel\Mcp\Server\Prompts\Argument>
     */
    public function arguments(): array
    {
        return [
            new Argument(
                name: 'tone',
                description: 'The tone to use in the weather description (e.g., formal, casual, humorous).',
                required: true,
            ),
        ];
    }
}
```

Validating Prompt Arguments

Prompt arguments are automatically validated based on their definition, but you may also want to enforce more complex validation rules.

Laravel MCP integrates seamlessly with Laravel's [validation features](#). You may validate incoming prompt arguments within your prompt's `handle` method:

```
<?php

namespace App\Mcp\Prompts;

use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Prompt;

class DescribeWeatherPrompt extends Prompt
{
    /**
     * Handle the prompt request.
     */
    public function handle(Request $request): Response
    {
        $validated = $request->validate([
            'tone' => 'required|string|max:50',
        ]);

        $tone = $validated['tone'];

        // Generate the prompt response using the given tone...
    }
}
```

On validation failure, AI clients will act based on the error messages you provide. As such, is critical to provide clear and actionable error messages:

```
$validated = $request->validate([
    'tone' => ['required', 'string', 'max:50'],
], [
    'tone.*' => 'You must specify a tone for the weather description. Examples include "formal", "casual", or "humorous".',
]);

```

Prompt Dependency Injection

The Laravel [service container](#) is used to resolve all prompts. As a result, you are able to type-hint any dependencies your prompt may need in its constructor. The declared dependencies will automatically be resolved and injected into the prompt instance:

```
<?php

namespace App\Mcp\Prompts;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Server\Prompt;

class DescribeWeatherPrompt extends Prompt
{
    /**
     * Create a new prompt instance.
     */
    public function __construct(
        protected WeatherRepository $weather,
    ) {}

    //
}
```

In addition to constructor injection, you may also type-hint dependencies in your prompt's `handle` method. The service container will automatically resolve and inject the dependencies when the method is called:

```
<?php

namespace App\Mcp\Prompts;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Prompt;

class DescribeWeatherPrompt extends Prompt
{
    /**
     * Handle the prompt request.
     */
    public function handle(Request $request, WeatherRepository $weather): Response
    {
        $isAvailable = $weather->isServiceAvailable();

        // ...
    }
}
```

Conditional Prompt Registration

You may conditionally register prompts at runtime by implementing the `shouldRegister` method in your prompt class. This method allows you to determine whether a prompt should be available based on application state, configuration, or request parameters:

```
<?php

namespace App\Mcp\Prompts;

use Laravel\Mcp\Request;
use Laravel\Mcp\Server\Prompt;

class CurrentWeatherPrompt extends Prompt
{
    /**
     * Determine if the prompt should be registered.
     */
    public function shouldRegister(Request $request): bool
    {
        return $request?->user()?->subscribed() ?? false;
    }
}
```

When a prompt's `shouldRegister` method returns `false`, it will not appear in the list of available prompts and cannot be invoked by AI clients.

Prompt Responses

Prompts may return a single `Laravel\Mcp\Response` or an iterable of `Laravel\Mcp\Response` instances. These responses encapsulate the content that will be sent to the AI client:

```
<?php

namespace App\Mcp\Prompts;

use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Prompt;

class DescribeWeatherPrompt extends Prompt
{
    /**
     * Handle the prompt request.
     *
     * @return array<int, \Laravel\Mcp\Response>
     */
    public function handle(Request $request): array
    {
        $tone = $request->string('tone');

        $systemMessage = "You are a helpful weather assistant. Please provide a weather
description in a {$tone} tone./";

        $userMessage = "What is the current weather like in New York City?";

        return [
            Response::text($systemMessage)->asAssistant(),
            Response::text($userMessage),
        ];
    }
}
```

You can use the `asAssistant()` method to indicate that a response message should be treated as coming from the AI assistant, while regular messages are treated as user input.

Resources

[Resources](#) enable your server to expose data and content that AI clients can read and use as context when interacting with language models. They provide a way to share static or dynamic information like documentation, configuration, or any data that helps inform AI responses.

Creating Resources

To create a resource, run the `make:mcp-resource` Artisan command:

```
php artisan make:mcp-resource WeatherGuidelinesResource
```

After creating a resource, register it in your server's `$resources` property:

```
<?php

namespace App\Mcp\Servers;

use App\Mcp\Resources\WeatherGuidelinesResource;
use Laravel\Mcp\Server;

class WeatherServer extends Server
{
    /**
     * The resources registered with this MCP server.
     *
     * @var array<int, class-string<\Laravel\Mcp\Server\Resource>>
     */
    protected array $resources = [
        WeatherGuidelinesResource::class,
    ];
}
```

Resource Name, Title, and Description

By default, the resource's name and title are derived from the class name. For example, `WeatherGuidelinesResource` will have a name of `weather-guidelines` and a title of `Weather Guidelines Resource`. You may customize these values by defining the `$name` and `$title` properties on your resource:

```
class WeatherGuidelinesResource extends Resource
{
    /**
     * The resource's name.
     */
    protected string $name = 'weather-api-docs';

    /**
     * The resource's title.
     */
    protected string $title = 'Weather API Documentation';

    // ...
}
```

Resource descriptions are not automatically generated. You should always provide a meaningful description by defining the `$description` property on your resource:

```
class WeatherGuidelinesResource extends Resource
{
    /**
     * The resource's description.
     */
    protected string $description = 'Comprehensive guidelines for using the Weather API.';

    //
}
```



The description is a critical part of the resource's metadata, as it helps AI models understand when and how to use the resource effectively.

Resource URI and MIME Type

Each resource is identified by a unique URI and has an associated MIME type that helps AI clients understand the resource's format.

By default, the resource's URI is generated based on the resource's name, so `WeatherGuidelinesResource` will have a URI of `weather://resources/weather-guidelines`. The default MIME type is `text/plain`.

You may customize these values by defining the `$uri` and `$mimeType` properties on your resource:

```
<?php

namespace App\Mcp\Resources;

use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * The resource's URI.
     */
    protected string $uri = 'weather://resources/guidelines';

    /**
     * The resource's MIME type.
     */
    protected string $mimeType = 'application/pdf';
}
```

The URI and MIME type help AI clients determine how to process and interpret the resource content appropriately.

Resource Request

Unlike tools and prompts, resources can not define input schemas or arguments. However, you can still interact with request object within your resource's `handle` method:

```
<?php

namespace App\Mcp\Resources;

use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * Handle the resource request.
     */
    public function handle(Request $request): Response
    {
        // ...
    }
}
```

Resource Dependency Injection

The Laravel [service container](#) is used to resolve all resources. As a result, you are able to type-hint any dependencies your resource may need in its constructor. The declared dependencies will automatically be resolved and injected into the resource instance:

```
<?php

namespace App\Mcp\Resources;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * Create a new resource instance.
     */
    public function __construct(
        protected WeatherRepository $weather,
    ) {}

    // ...
}
```

In addition to constructor injection, you may also type-hint dependencies in your resource's `handle` method. The service container will automatically resolve and inject the dependencies when the method is called:

```
<?php

namespace App\Mcp\Resources;

use App\Repositories\WeatherRepository;
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;
use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * Handle the resource request.
     */
    public function handle(WeatherRepository $weather): Response
    {
        $guidelines = $weather->guidelines();

        return Response::text($guidelines);
    }
}
```

Conditional Resource Registration

You may conditionally register resources at runtime by implementing the `shouldRegister` method in your resource class. This method allows you to determine whether a resource should be available based on application state, configuration, or request parameters:

```
<?php

namespace App\Mcp\Resources;

use Laravel\Mcp\Request;
use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * Determine if the resource should be registered.
     */
    public function shouldRegister(Request $request): bool
    {
        return $request?->user()?->subscribed() ?? false;
    }
}
```

When a resource's `shouldRegister` method returns `false`, it will not appear in the list of available resources and cannot be accessed by AI clients.

Resource Responses

Resources must return an instance of `Laravel\Mcp\Response`. The Response class provides several convenient methods for creating different types of responses:

For simple text content, use the `text` method:

```
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;

/**
 * Handle the resource request.
 */
public function handle(Request $request): Response
{
    // ...

    return Response::text($weatherData);
}
```

Blob Responses

To return blob content, use the `blob` method, providing the blob content:

```
return Response::blob(file_get_contents(storage_path('weather/radar.png')));
```

When returning blob content, the MIME type will be determined by the value of the `$mimeType` property on the resource class:

```
<?php

namespace App\Mcp\Resources;

use Laravel\Mcp\Server\Resource;

class WeatherGuidelinesResource extends Resource
{
    /**
     * The resource's MIME type.
     */
    protected string $mimeType = 'image/png';

    //
}
```

Error Responses

To indicate an error occurred during resource retrieval, use the `error()` method:

```
return Response::error('Unable to fetch weather data for the specified location.');
```

Authentication

You can authenticate web MCP servers with middleware just like you would for routes. This will require a user to authenticate before using any capability of the server.

There are two ways to authenticate access to your MCP server: simple, token based authentication via [Laravel Sanctum](#), or any other arbitrary API tokens which are passed via the `Authorization` HTTP header. Or, you may authenticate via OAuth using [Laravel Passport](#).

OAuth 2.1

The most robust way to protect your web-based MCP servers is with OAuth through [Laravel Passport](#).

When authenticating your MCP server via OAuth, you will invoke the `Mcp::oauthRoutes` method in your `routes/api.php` file to register the required OAuth2 discovery and client registration routes. Then, apply Passport's `auth:api` middleware to your `Mcp::web` route in your `routes/api.php` file:

```
use App\Mcp\Servers\WeatherExample;
use Laravel\Mcp\Facades\Mcp;

Mcp::oauthRoutes();

Mcp::web('/mcp/weather', WeatherExample::class)
    ->middleware('auth:api');
```

New Passport Installation

If your application is not already using Laravel Passport, start by following Passport's [installation and deployment steps](#). You should have an `OAuthAuthenticatable` model, new authentication guard, and passport keys before moving on.

Next, you should publish Laravel MCP's provided Passport authorization view:

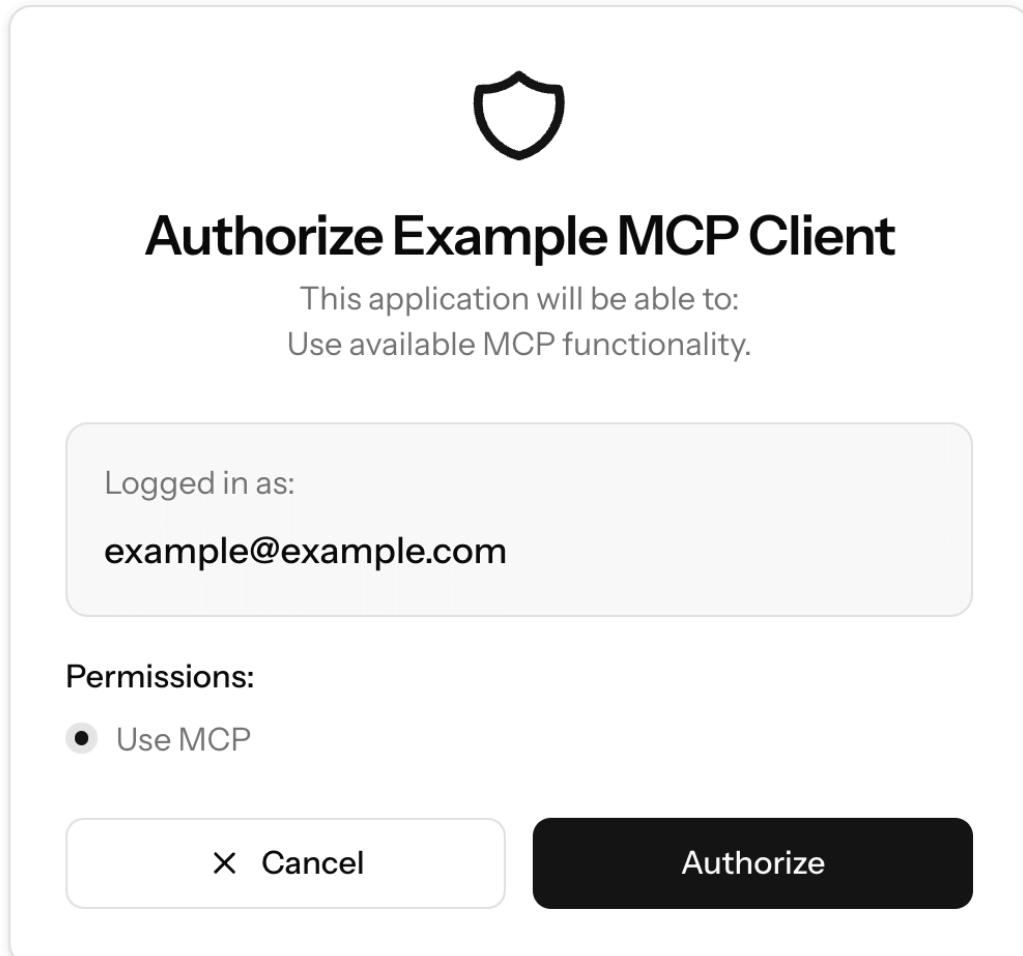
```
php artisan vendor:publish --tag=mcp-views
```

Then, instruct Passport to use this view using the `Passport::authorizationView` method. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider`:

```
use Laravel\Passport\Passport;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Passport::authorizationView(function ($parameters) {
        return view('mcp.authorize', $parameters);
    });
}
```

This view will be displayed to the end-user during authentication to reject or approve the AI agent's authentication attempt.



In this scenario, we're simply using OAuth as a translation layer to the underlying authenticatable model. We are ignoring many aspects of OAuth, such as scopes.

Using an Existing Passport Installation

If your application is already using Laravel Passport, Laravel MCP should work seamlessly within your existing Passport installation, but custom scopes aren't currently supported as OAuth is primarily used as a translation layer to the underlying authenticatable model.

Laravel MCP, via the `Mcp::oauthRoutes()` method discussed above, adds, advertises, and uses a single `mcp:use` scope.

Passport vs. Sanctum

OAuth2.1 is the documented authentication mechanism in the Model Context Protocol specification, and is the most widely supported among MCP clients. For that reason, we recommend using Passport when possible.

If your application is already using Sanctum then adding Passport may be cumbersome. In this instance, we recommend using Sanctum without Passport until you have a clear, necessary requirement to use an MCP client that only supports OAuth.

Sanctum

If you would like to protect your MCP server using [Sanctum](#), simply add Sanctum's authentication middleware to your server in your `routes/ai.php` file. Then, ensure your MCP clients provide a `Authorization: Bearer <token>` header to ensure successful authentication:

```
use App\Mcp\Servers\WeatherExample;
use Laravel\Mcp\Facades\Mcp;

Mcp::web('/mcp/demo', WeatherExample::class)
    ->middleware('auth:sanctum');
```

Custom MCP Authentication

If your application issues its own custom API tokens, you may authenticate your MCP server by assigning any middleware you wish to your `Mcp::web` routes. Your custom middleware can inspect the `Authorization` header manually to authenticate the incoming MCP request.

Authorization

You may access the currently authenticated user via the `$request->user()` method, allowing you to perform [authorization checks](#) within your MCP tools and resources:

```
use Laravel\Mcp\Request;
use Laravel\Mcp\Response;

/**
 * Handle the tool request.
 */
public function handle(Request $request): Response
{
    if (! $request->user()->can('read-weather')) {
        return Response::error('Permission denied.');
    }

    // ...
}
```

Testing Servers

You may test your MCP servers using the built-in MCP Inspector or by writing unit tests.

MCP Inspector

The [MCP Inspector](#) is an interactive tool for testing and debugging your MCP servers. Use it to connect to your server, verify authentication, and try out tools, resources, and prompts.

You may run the inspector for any registered server (for example, a local server named "weather"):

```
php artisan mcp:inspector weather
```

This command launches the MCP Inspector and provides the client settings that you may copy into your MCP client to ensure everything is configured correctly. If your web server is protected by an authentication middleware, make sure to include the required headers, such as an `Authorization` bearer token, when connecting.

Unit Tests

You may write unit tests for your MCP servers, tools, resources, and prompts.

To get started, create a new test case and invoke the desired primitive on the server that registers it. For example, to test a tool on the `WeatherServer`:

```
test('tool', function () {
    $response = WeatherServer::tool(WeatherTool::class, [
        'location' => 'New York City',
        'units' => 'fahrenheit',
    ]);

    $response
        ->assertOk()
        ->assertSee('The current weather in New York City is 72°F and sunny.');
});
```

```
/*
 * Test a tool.
 */
public function test_tool(): void
{
    $response = WeatherServer::tool(WeatherTool::class, [
        'location' => 'New York City',
        'units' => 'fahrenheit',
    ]);

    $response
        ->assertOk()
        ->assertSee('The current weather in New York City is 72°F and sunny.');
}
```

Similarly, you may test prompts and resources:

```
$response = WeatherServer::prompt(...);
$response = WeatherServer::resource(...);
```

You may also act as an authenticated user by chaining the `actingAs` method before invoking the primitive:

```
$response = WeatherServer::actingAs($user)->tool(...);
```

Once you receive the response, you may use various assertion methods to verify the content and status of the response.

You may assert that a response is successful using the `assertOk` method. This checks that the response does not have any errors:

```
$response->assertOk();
```

You may assert that a response contains specific text using the `assertSee` method:

```
$response->assertSee('The current weather in New York City is 72°F and sunny.');
```

You may assert that a response contains an error using the `assertHasErrors()` method:

```
$response->assertHasErrors();  
  
$response->assertHasErrors([
    'Something went wrong.',
]);
```

You may assert that a response does not contain an error using the `assertHasNoErrors()` method:

```
$response->assertHasNoErrors();
```

You may assert that a response contains specific metadata using the `assertName()`, `assertTitle()`, and `assertDescription()` methods:

```
$response->assertName('current-weather');
$response->assertTitle('Current Weather Tool');
$response->assertDescription('Fetches the current weather forecast for a specified location.');
```

You may assert that notifications were sent using the `assertSentNotification()` and `assertNotificationCount()` methods:

```
$response->assertSentNotification('processing/progress', [
    'step' => 1,
    'total' => 5,
]);  
  
$response->assertSentNotification('processing/progress', [
    'step' => 2,
    'total' => 5,
]);  
  
$response->assertNotificationCount(5);
```

Finally, if you wish to inspect the raw response content, you may use the `dd()` or `dump()` methods to output the response for debugging purposes:

```
$response->dd();
$response->dump();
```

Laravel Mix

- [Introduction](#)

Introduction

[Laravel Mix](#), a package developed by [Laracasts](#) creator Jeffrey Way, provides a fluent API for defining [webpack](#) build steps for your Laravel application using several common CSS and JavaScript pre-processors.

In other words, Mix makes it a cinch to compile and minify your application's CSS and JavaScript files. Through simple method chaining, you can fluently define your asset pipeline. For example:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css');
```

If you've ever been confused and overwhelmed about getting started with webpack and asset compilation, you will love Laravel Mix. However, you are not required to use it while developing your application; you are free to use any asset pipeline tool you wish, or even none at all.



Vite has replaced Laravel Mix in new Laravel installations. For Mix documentation, please visit the [official Laravel Mix website](#). If you would like to switch to Vite, please see our [Vite migration guide](#).

Laravel Octane

- [Introduction](#)
- [Installation](#)
- [Server Prerequisites](#)
 - [FrankenPHP](#)
 - [RoadRunner](#)
 - [Swoole](#)
- [Serving Your Application](#)
 - [Serving Your Application via HTTPS](#)
 - [Serving Your Application via Nginx](#)
 - [Watching for File Changes](#)
 - [Specifying the Worker Count](#)
 - [Specifying the Max Request Count](#)
 - [Reloading the Workers](#)
 - [Stopping the Server](#)
- [Dependency Injection and Octane](#)
 - [Container Injection](#)
 - [Request Injection](#)
 - [Configuration Repository Injection](#)
- [Managing Memory Leaks](#)
- [Concurrent Tasks](#)
- [Ticks and Intervals](#)
- [The Octane Cache](#)
- [Tables](#)

Introduction

[Laravel Octane](#) supercharges your application's performance by serving your application using high-powered application servers, including [FrankenPHP](#), [Open Swoole](#), [Swoole](#), and [RoadRunner](#). Octane boots your application once, keeps it in memory, and then feeds it requests at supersonic speeds.

Installation

Octane may be installed via the Composer package manager:

```
composer require laravel/octane
```

After installing Octane, you may execute the `octane:install` Artisan command, which will install Octane's configuration file into your application:

```
php artisan octane:install
```

Server Prerequisites



Laravel Octane requires [PHP 8.1+](#).

FrankenPHP



FrankenPHP's Octane integration is in beta and should be used with caution in production.

FrankenPHP is a PHP application server, written in Go, that supports modern web features like early hints and Zstandard compression. When you install Octane and choose FrankenPHP as your server, Octane will automatically download and install the FrankenPHP binary for you.

FrankenPHP via Laravel Sail

If you plan to develop your application using Laravel Sail, you should run the following commands to install Octane and FrankenPHP:

```
./vendor/bin/sail up
./vendor/bin/sail composer require laravel/octane
```

Next, you should use the `octane:install` Artisan command to install the FrankenPHP binary:

```
./vendor/bin/sail artisan octane:install --server=frankenphp
```

Finally, add a `SUPERVISOR_PHP_COMMAND` environment variable to the `laravel.test` service definition in your application's `docker-compose.yml` file. This environment variable will contain the command that Sail will use to serve your application using Octane instead of the PHP development server:

```
services:
  laravel.test:
    environment:
      + SUPERVISOR_PHP_COMMAND: "/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
        octane:start --server=fankenphp --host=0.0.0.0 --admin-port=2019 --port=80"
      + XDG_CONFIG_HOME: /var/www/html/config
      + XDG_DATA_HOME: /var/www/html/data
```

To enable HTTPS, HTTP/2, and HTTP/3, apply these modifications instead:

```
services:
  laravel.test:
    ports:
      - '${APP_PORT:-80}:80'
      - '${VITE_PORT:-5173}:${VITE_PORT:-5173}'
    environment:
      + SUPERVISOR_PHP_COMMAND: "/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
        octane:start --host=localhost --port=443 --admin-port=2019 --https"
      + XDG_CONFIG_HOME: /var/www/html/config
      + XDG_DATA_HOME: /var/www/html/data
```

Typically, you should access your FrankenPHP Sail application via `https://localhost`, as using `https://127.0.0.1` requires additional configuration and is discouraged.

FrankenPHP via Docker

Using FrankenPHP's official Docker images can offer improved performance and the use additional extensions not included with static installations of FrankenPHP. In addition, the official Docker images provide support for running FrankenPHP on platforms it doesn't natively support, such as Windows. FrankenPHP's official Docker images are suitable for both local development and production usage.

You may use the following Dockerfile as a starting point for containerizing your FrankenPHP powered Laravel application:

```
FROM dunglas/frankenphp

RUN install-php-extensions \
    pcntl
    # Add other PHP extensions here...

COPY . /app

ENTRYPOINT ["php", "artisan", "octane:frankenphp"]
```

Then, during development, you may utilize the following Docker Compose file to run your application:

```
# compose.yaml
services:
  frankenphp:
    build:
      context: .
    entrypoint: php artisan octane:frankenphp --max-requests=1
    ports:
      - "8000:8000"
    volumes:
      - .:/app
```

You may consult [the official FrankenPHP documentation](#) for more information on running FrankenPHP with Docker.

RoadRunner

[RoadRunner](#) is powered by the RoadRunner binary, which is built using Go. The first time you start a RoadRunner based Octane server, Octane will offer to download and install the RoadRunner binary for you.

RoadRunner via Laravel Sail

If you plan to develop your application using [Laravel Sail](#), you should run the following commands to install Octane and RoadRunner:

```
./vendor/bin/sail up

./vendor/bin/sail composer require laravel/octane spiral/roadrunner-cli spiral/roadrunner-http
```

Next, you should start a Sail shell and use the `rr` executable to retrieve the latest Linux based build of the RoadRunner binary:

```
./vendor/bin/sail shell

# Within the Sail shell...
./vendor/bin/rr get-binary
```

Then, add a `SUPERVISOR_PHP_COMMAND` environment variable to the `laravel.test` service definition in your application's `docker-compose.yml` file. This environment variable will contain the command that Sail will use to serve your application using Octane instead of the PHP development server:

```
services:
  laravel.test:
    environment:
      + SUPERVISOR_PHP_COMMAND: "/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
        octane:start --server=roadrunner --host=0.0.0.0 --rpc-port=6001 --port=80"
```

Finally, ensure the `rr` binary is executable and build your Sail images:

```
chmod +x ./rr
./vendor/bin/sail build --no-cache
```

Swoole

If you plan to use the Swoole application server to serve your Laravel Octane application, you must install the Swoole PHP extension. Typically, this can be done via PECL:

```
pecl install swoole
```

Open Swoole

If you want to use the Open Swoole application server to serve your Laravel Octane application, you must install the Open Swoole PHP extension. Typically, this can be done via PECL:

```
pecl install openswoole
```

Using Laravel Octane with Open Swoole grants the same functionality provided by Swoole, such as concurrent tasks, ticks, and intervals.

Swoole via Laravel Sail



Before serving an Octane application via Sail, ensure you have the latest version of Laravel Sail and execute `./vendor/bin/sail build --no-cache` within your application's root directory.

Alternatively, you may develop your Swoole based Octane application using [Laravel Sail](#), the official Docker based development environment for Laravel. Laravel Sail includes the Swoole extension by default. However, you will still need to adjust the `docker-compose.yml` file used by Sail.

To get started, add a `SUPERVISOR_PHP_COMMAND` environment variable to the `laravel.test` service definition in your application's `docker-compose.yml` file. This environment variable will contain the command that Sail will use to serve your application using Octane instead of the PHP development server:

```
services:
  laravel.test:
    environment:
      + SUPERVISOR_PHP_COMMAND: "/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
        octane:start --server=swoole --host=0.0.0.0 --port=80"
```

Finally, build your Sail images:

```
./vendor/bin/sail build --no-cache
```

Swoole Configuration

Swoole supports a few additional configuration options that you may add to your `octane` configuration file if necessary. Because they rarely need to be modified, these options are not included in the default configuration file:

```
'swoole' => [
  'options' => [
    'log_file' => storage_path('logs/swoole_http.log'),
    'package_max_length' => 10 * 1024 * 1024,
  ],
],
```

Serving Your Application

The Octane server can be started via the `octane:start` Artisan command. By default, this command will utilize the server specified by the `server` configuration option of your application's `octane` configuration file:

```
php artisan octane:start
```

By default, Octane will start the server on port 8000, so you may access your application in a web browser via `http://localhost:8000`.

Serving Your Application via HTTPS

By default, applications running via Octane generate links prefixed with `http://`. The `OCTANE_HTTPS` environment variable, used within your application's `config/octane.php` configuration file, can be set to `true` when serving your application via HTTPS. When this configuration value is set to `true`, Octane will instruct Laravel to prefix all generated links with `https://`:

```
'https' => env('OCTANE_HTTPS', false),
```

Serving Your Application via Nginx



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 Octane application, check out [Laravel Forge](#).

In production environments, you should serve your Octane application behind a traditional web server such as Nginx or Apache. Doing so will allow the web server to serve your static assets such as images and stylesheets, as well as manage your SSL certificate termination.

In the Nginx configuration example below, Nginx will serve the site's static assets and proxy requests to the Octane server that is running on port 8000:

```

map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

server {
    listen 80;
    listen [::]:80;
    server_name domain.com;
    server_tokens off;
    root /home/forge/domain.com/public;

    index index.php;

    charset utf-8;

    location /index.php {
        try_files /not_exists @octane;
    }

    location / {
        try_files $uri $uri/ @octane;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt { access_log off; log_not_found off; }

    access_log off;
    error_log  /var/log/nginx/domain.com-error.log error;

    error_page 404 /index.php;

    location @octane {
        set $suffix "";
        if ($uri = /index.php) {
            set $suffix ?$query_string;
        }

        proxy_http_version 1.1;
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header SERVER_PORT $server_port;
        proxy_set_header REMOTE_ADDR $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

        proxy_pass http://127.0.0.1:8000$suffix;
    }
}

```

Watching for File Changes

Since your application is loaded in memory once when the Octane server starts, any changes to your application's files will not be reflected when you refresh your browser. For example, route definitions added to your `routes/web.php` file will not be reflected until the server is restarted. For convenience, you may use the `--watch` flag to instruct Octane to automatically restart the server on any file changes within your application:

```
php artisan octane:start --watch
```

Before using this feature, you should ensure that [Node](#) is installed within your local development environment. In addition, you should install the [Chokidar](#) file-watching library within your project:

```
npm install --save-dev chokidar
```

You may configure the directories and files that should be watched using the `watch` configuration option within your application's `config/octane.php` configuration file.

Specifying the Worker Count

By default, Octane will start an application request worker for each CPU core provided by your machine. These workers will then be used to serve incoming HTTP requests as they enter your application. You may manually specify how many workers you would like to start using the `--workers` option when invoking the `octane:start` command:

```
php artisan octane:start --workers=4
```

If you are using the Swoole application server, you may also specify how many ["task workers"](#) you wish to start:

```
php artisan octane:start --workers=4 --task-workers=6
```

Specifying the Max Request Count

To help prevent stray memory leaks, Octane gracefully restarts any worker once it has handled 500 requests. To adjust this number, you may use the `--max-requests` option:

```
php artisan octane:start --max-requests=250
```

Reloading the Workers

You may gracefully restart the Octane server's application workers using the `octane:reload` command. Typically, this should be done after deployment so that your newly deployed code is loaded into memory and is used to serve to subsequent requests:

```
php artisan octane:reload
```

Stopping the Server

You may stop the Octane server using the `octane:stop` Artisan command:

```
php artisan octane:stop
```

Checking the Server Status

You may check the current status of the Octane server using the `octane:status` Artisan command:

```
php artisan octane:status
```

Dependency Injection and Octane

Since Octane boots your application once and keeps it in memory while serving requests, there are a few caveats you should consider while building your application. For example, the `register` and `boot` methods of your application's service providers will only be executed once when the request worker initially boots. On subsequent requests, the same application instance will be reused.

In light of this, you should take special care when injecting the application service container or request into any object's constructor. By doing so, that object may have a stale version of the container or request on subsequent requests.

Octane will automatically handle resetting any first-party framework state between requests. However, Octane does not always know how to reset the global state created by your application. Therefore, you should be aware of how to build your application in a way that is Octane friendly. Below, we will discuss the most common situations that may cause problems while using Octane.

Container Injection

In general, you should avoid injecting the application service container or HTTP request instance into the constructors of other objects. For example, the following binding injects the entire application service container into an object that is bound as a singleton:

```
use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app);
    });
}
```

In this example, if the `Service` instance is resolved during the application boot process, the container will be injected into the service and that same container will be held by the `Service` instance on subsequent requests. This **may** not be a problem for your particular application; however, it can lead to the container unexpectedly missing bindings that were added later in the boot cycle or by a subsequent request.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a container resolver closure into the service that always resolves the current container instance:

```

use App\Service;
use Illuminate\Container\Container;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app);
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () => Container::getInstance());
});

```

The global `app` helper and the `Container::getInstance()` method will always return the latest version of the application container.

Request Injection

In general, you should avoid injecting the application service container or HTTP request instance into the constructors of other objects. For example, the following binding injects the entire request instance into an object that is bound as a singleton:

```

use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app['request']);
    });
}

```

In this example, if the `Service` instance is resolved during the application boot process, the HTTP request will be injected into the service and that same request will be held by the `Service` instance on subsequent requests. Therefore, all headers, input, and query string data will be incorrect, as well as all other request data.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a request resolver closure into the service that always resolves the current request instance. Or, the most recommended approach is simply to pass the specific request information your object needs to one of the object's methods at runtime:

```

use App\Service;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app['request']);
});

$this->app->singleton(Service::class, function (Application $app) {
    return new Service(fn () => $app['request']);
});

// Or...

$service->method($request->input('name'));

```

The global `request` helper will always return the request the application is currently handling and is therefore safe to use within your application.



It is acceptable to type-hint the `Illuminate\Http\Request` instance on your controller methods and route closures.

Configuration Repository Injection

In general, you should avoid injecting the configuration repository instance into the constructors of other objects. For example, the following binding injects the configuration repository into an object that is bound as a singleton:

```

use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app->make('config'));
    });
}

```

In this example, if the configuration values change between requests, that service will not have access to the new values because it's depending on the original repository instance.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a configuration repository resolver closure to the class:

```

use App\Service;
use Illuminate\Container\Container;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app->make('config'));
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () => Container::getInstance()->make('config'));
});

```

The global `config` will always return the latest version of the configuration repository and is therefore safe to use within your application.

Managing Memory Leaks

Remember, Octane keeps your application in memory between requests; therefore, adding data to a statically maintained array will result in a memory leak. For example, the following controller has a memory leak since each request to the application will continue to add data to the static `$data` array:

```

use App\Service;
use Illuminate\Http\Request;
use Illuminate\Support\Str;

/**
 * Handle an incoming request.
 */
public function index(Request $request): array
{
    Service::$data[] = Str::random(10);

    return [
        // ...
    ];
}

```

While building your application, you should take special care to avoid creating these types of memory leaks. It is recommended that you monitor your application's memory usage during local development to ensure you are not introducing new memory leaks into your application.

Concurrent Tasks



This feature requires [Swoole](#).

When using Swoole, you may execute operations concurrently via light-weight background tasks. You may accomplish this using Octane's `concurrently` method. You may combine this method with PHP array destructuring to retrieve the results of each operation:

```
use App\Models\User;
use App\Models\Server;
use Laravel\Octane\Facades\Octane;

[$users, $servers] = Octane::concurrently([
    fn () => User::all(),
    fn () => Server::all(),
]);
```

Concurrent tasks processed by Octane utilize Swoole's "task workers", and execute within an entirely different process than the incoming request. The amount of workers available to process concurrent tasks is determined by the `--task-workers` directive on the `octane:start` command:

```
php artisan octane:start --workers=4 --task-workers=6
```

When invoking the `concurrently` method, you should not provide more than 1024 tasks due to limitations imposed by Swoole's task system.

Ticks and Intervals



This feature requires [Swoole](#).

When using Swoole, you may register "tick" operations that will be executed every specified number of seconds. You may register "tick" callbacks via the `tick` method. The first argument provided to the `tick` method should be a string that represents the name of the ticker. The second argument should be a callable that will be invoked at the specified interval.

In this example, we will register a closure to be invoked every 10 seconds. Typically, the `tick` method should be called within the `boot` method of one of your application's service providers:

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
->seconds(10);
```

Using the `immediate` method, you may instruct Octane to immediately invoke the tick callback when the Octane server initially boots, and every N seconds thereafter:

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
->seconds(10)
->immediate();
```

The Octane Cache



This feature requires [Swoole](#).

When using Swoole, you may leverage the Octane cache driver, which provides read and write speeds of up to 2 million operations per second. Therefore, this cache driver is an excellent choice for applications that need extreme read / write speeds from their caching layer.

This cache driver is powered by [Swoole tables](#). All data stored in the cache is available to all workers on the server. However, the cached data will be flushed when the server is restarted:

```
Cache::store('octane')->put('framework', 'Laravel', 30);
```



The maximum number of entries allowed in the Octane cache may be defined in your application's `octane` configuration file.

Cache Intervals

In addition to the typical methods provided by Laravel's cache system, the Octane cache driver features interval based caches. These caches are automatically refreshed at the specified interval and should be registered within the `boot` method of one of your application's service providers. For example, the following cache will be refreshed every five seconds:

```
use Illuminate\Support\Str;

Cache::store('octane')->interval('random', function () {
    return Str::random(10);
}, seconds: 5);
```

Tables



This feature requires [Swoole](#).

When using Swoole, you may define and interact with your own arbitrary [Swoole tables](#). Swoole tables provide extreme performance throughput and the data in these tables can be accessed by all workers on the server. However, the data within them will be lost when the server is restarted.

Tables should be defined within the `tables` configuration array of your application's `octane` configuration file. An example table that allows a maximum of 1000 rows is already configured for you. The maximum size of string columns may be configured by specifying the column size after the column type as seen below:

```
'tables' => [
    'example:1000' => [
        'name' => 'string:1000',
        'votes' => 'int',
    ],
],
```

To access a table, you may use the `Octane::table` method:

```
use Laravel\Octane\Facades\Octane;

Octane::table('example')->set('uuid', [
    'name' => 'Nuno Maduro',
    'votes' => 1000,
]);

return Octane::table('example')->get('uuid');
```



The column types supported by Swoole tables are: `string`, `int`, and `float`.

Laravel Passport

- [Introduction](#)
 - [Passport or Sanctum?](#)
- [Installation](#)
 - [Deploying Passport](#)
 - [Migration Customization](#)
 - [Upgrading Passport](#)
- [Configuration](#)
 - [Client Secret Hashing](#)
 - [Token Lifetimes](#)
 - [Overriding Default Models](#)
 - [Overriding Routes](#)
- [Issuing Access Tokens](#)
 - [Managing Clients](#)
 - [Requesting Tokens](#)
 - [Refreshing Tokens](#)
 - [Revoking Tokens](#)
 - [Purging Tokens](#)
- [Authorization Code Grant With PKCE](#)
 - [Creating the Client](#)
 - [Requesting Tokens](#)
- [Password Grant Tokens](#)
 - [Creating a Password Grant Client](#)
 - [Requesting Tokens](#)
 - [Requesting All Scopes](#)
 - [Customizing the User Provider](#)
 - [Customizing the Username Field](#)
 - [Customizing the Password Validation](#)
- [Implicit Grant Tokens](#)
- [Client Credentials Grant Tokens](#)
- [Personal Access Tokens](#)
 - [Creating a Personal Access Client](#)
 - [Managing Personal Access Tokens](#)
- [Protecting Routes](#)
 - [Via Middleware](#)
 - [Passing the Access Token](#)
- [Token Scopes](#)
 - [Defining Scopes](#)
 - [Default Scope](#)
 - [Assigning Scopes to Tokens](#)
 - [Checking Scopes](#)
- [Consuming Your API With JavaScript](#)
- [Events](#)
- [Testing](#)

Introduction

[Laravel Passport](#) provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the [League OAuth2 server](#) that is maintained by Andy Millington and Simon Hamp.



This documentation assumes you are already familiar with OAuth2. If you do not know anything about OAuth2, consider familiarizing yourself with the general [terminology](#) and features of OAuth2 before continuing.

Passport or Sanctum?

Before getting started, you may wish to determine if your application would be better served by Laravel Passport or [Laravel Sanctum](#). If your application absolutely needs to support OAuth2, then you should use Laravel Passport.

However, if you are attempting to authenticate a single-page application, mobile application, or issue API tokens, you should use [Laravel Sanctum](#). Laravel Sanctum does not support OAuth2; however, it provides a much simpler API authentication development experience.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Passport's [service provider](#) registers its own database migration directory, so you should migrate your database after installing the package. The Passport migrations will create the tables your application needs to store OAuth2 clients and access tokens:

```
php artisan migrate
```

Next, you should execute the `passport:install` Artisan command. This command will create the encryption keys needed to generate secure access tokens. In addition, the command will create "personal access" and "password grant" clients which will be used to generate access tokens:

```
php artisan passport:install
```



If you would like to use UUIDs as the primary key value of the Passport `Client` model instead of auto-incrementing integers, please install Passport using the [uids option](#).

After running the `passport:install` command, add the `Laravel\Passport\HasApiTokens` trait to your `App\Models\User` model. This trait will provide a few helper methods to your model which allow you to inspect the authenticated user's token and scopes. If your model is already using the `Laravel\Sanctum\HasApiTokens` trait, you may remove that trait:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

Finally, in your application's `config/auth.php` configuration file, you should define an `api` authentication guard and set the `driver` option to `passport`. This will instruct your application to use Passport's `TokenGuard` when authenticating incoming API requests:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Client UUIDs

You may also run the `passport:install` command with the `--uuids` option present. This option will instruct Passport that you would like to use UUIDs instead of auto-incrementing integers as the Passport `Client` model's primary key values. After running the `passport:install` command with the `--uuids` option, you will be given additional instructions regarding disabling Passport's default migrations:

```
php artisan passport:install --uuids
```

Deploying Passport

When deploying Passport to your application's servers for the first time, you will likely need to run the `passport:keys` command. This command generates the encryption keys Passport needs in order to generate access tokens. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

If necessary, you may define the path where Passport's keys should be loaded from. You may use the `Passport::loadKeysFrom` method to accomplish this. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::loadKeysFrom(__DIR__ . '/../secrets/oauth');
}
```

Loading Keys From the Environment

Alternatively, you may publish Passport's configuration file using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-config
```

After the configuration file has been published, you may load your application's encryption keys by defining them as environment variables:

```
PASSPORT_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
<private key here>
-----END RSA PRIVATE KEY-----"

PASSPORT_PUBLIC_KEY="-----BEGIN PUBLIC KEY-----
<public key here>
-----END PUBLIC KEY-----"
```

Migration Customization

If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-migrations
```

Upgrading Passport

When upgrading to a new major version of Passport, it's important that you carefully review [the upgrade guide](#).

Configuration

Client Secret Hashing

If you would like your client's secrets to be hashed when stored in your database, you should call the `Passport::hashClientSecrets` method in the `boot` method of your `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

Passport::hashClientSecrets();
```

Once enabled, all of your client secrets will only be displayable to the user immediately after they are created. Since the plain-text client secret value is never stored in the database, it is not possible to recover the secret's value if it is lost.

Token Lifetimes

By default, Passport issues long-lived access tokens that expire after one year. If you would like to configure a longer / shorter token lifetime, you may use the `tokensExpireIn`, `refreshTokensExpireIn`, and `personalAccessTokensExpireIn` methods. These methods should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::tokensExpireIn(now()>addDays(15));
    Passport::refreshTokensExpireIn(now()>addDays(30));
    Passport::personalAccessTokensExpireIn(now()>addMonths(6));
}
```



The `expires_at` columns on Passport's database tables are read-only and for display purposes only. When issuing tokens, Passport stores the expiration information within the signed and encrypted tokens. If you need to invalidate a token you should [revoke it](#).

Overriding Default Models

You are free to extend the models used internally by Passport by defining your own model and extending the corresponding Passport model:

```
use Laravel\Passport\Client as PassportClient;

class Client extends PassportClient
{
    // ...
}
```

After defining your model, you may instruct Passport to use your custom model via the `Laravel\Passport\Passport` class. Typically, you should inform Passport about your custom models in the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```

use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessClient;
use App\Models\Passport\RefreshToken;
use App\Models\Passport\Token;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::useTokenModel(Token::class);
    Passport::useRefreshTokenModel(RefreshToken::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::useClientModel(Client::class);
    Passport::usePersonalAccessClientModel(PersonalAccessClient::class);
}

```

Overriding Routes

Sometimes you may wish to customize the routes defined by Passport. To achieve this, you first need to ignore the routes registered by Passport by adding `Passport::ignoreRoutes` to the `register` method of your application's `AppServiceProvider`:

```

use Laravel\Passport\Passport;

/**
 * Register any application services.
 */
public function register(): void
{
    Passport::ignoreRoutes();
}

```

Then, you may copy the routes defined by Passport in [its routes file](#) to your application's `routes/web.php` file and modify them to your liking:

```

Route::group([
    'as' => 'passport.',
    'prefix' => config('passport.path', 'oauth'),
    'namespace' => '\Laravel\Passport\Http\Controllers',
], function () {
    // Passport routes...
});

```

Issuing Access Tokens

Using OAuth2 via authorization codes is how most developers are familiar with OAuth2. When using authorization codes, a client application will redirect a user to your server where they will either approve or deny the request to issue an access token to the client.

Managing Clients

First, developers building applications that need to interact with your application's API will need to register their application with yours by creating a "client". Typically, this consists of providing the name of their application and a URL that your application can redirect to after users approve their request for authorization.

The `passport:client` Command

The simplest way to create a client is using the `passport:client` Artisan command. This command may be used to create your own clients for testing your OAuth2 functionality. When you run the `client` command, Passport will prompt you for more information about your client and will provide you with a client ID and secret:

```
php artisan passport:client
```

Redirect URLs

If you would like to allow multiple redirect URLs for your client, you may specify them using a comma-delimited list when prompted for the URL by the `passport:client` command. Any URLs which contain commas should be URL encoded:

```
http://example.com/callback,http://examplefoo.com/callback
```

JSON API

Since your application's users will not be able to utilize the `client` command, Passport provides a JSON API that you may use to create clients. This saves you the trouble of having to manually code controllers for creating, updating, and deleting clients.

However, you will need to pair Passport's JSON API with your own frontend to provide a dashboard for your users to manage their clients. Below, we'll review all of the API endpoints for managing clients. For convenience, we'll use [Axios](#) to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

`GET /oauth/clients`

This route returns all of the clients for the authenticated user. This is primarily useful for listing all of the user's clients so that they may edit or delete them:

```
axios.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
 });
```

`POST /oauth/clients`

This route is used to create new clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization.

When a client is created, it will be issued a client ID and client secret. These values will be used when requesting access tokens from your application. The client creation route will return the new client instance:

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });

```

PUT /oauth/clients/{client-id}

This route is used to update clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization. The route will return the updated client instance:

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });

```

DELETE /oauth/clients/{client-id}

This route is used to delete clients:

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        // ...
    });

```

Requesting Tokens

Redirecting for Authorization

Once a client has been created, developers may use their client ID and secret to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```

use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});

```

The `prompt` parameter may be used to specify the authentication behavior of the Passport application.

If the `prompt` value is `none`, Passport will always throw an authentication error if the user is not already authenticated with the Passport application. If the value is `consent`, Passport will always display the authorization approval screen, even if all scopes were previously granted to the consuming application. When the value is `login`, the Passport application will always prompt the user to re-login to the application, even if they already have an existing session.

If no `prompt` value is provided, the user will be prompted for authorization only if they have not previously authorized access to the consuming application for the requested scopes.



Remember, the `/oauth/authorize` route is already defined by Passport. You do not need to manually define this route.

Approving the Request

When receiving authorization requests, Passport will automatically respond based on the value of `prompt` parameter (if present) and may display a template to the user allowing them to approve or deny the authorization request. If they approve the request, they will be redirected back to the `redirect_uri` that was specified by the consuming application. The `redirect_uri` must match the `redirect` URL that was specified when the client was created.

If you would like to customize the authorization approval screen, you may publish Passport's views using the `vendor:publish` Artisan command. The published views will be placed in the `resources/views/vendor/passport` directory:

```
php artisan vendor:publish --tag=passport-views
```

Sometimes you may wish to skip the authorization prompt, such as when authorizing a first-party client. You may accomplish this by [extending the `Client` model](#) and defining a `skipsAuthorization` method. If `skipsAuthorization` returns `true` the client will be approved and the user will be redirected back to the `redirect_uri` immediately, unless the consuming application has explicitly set the `prompt` parameter when redirecting for authorization:

```
<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;

class Client extends BaseClient
{
    /**
     * Determine if the client should skip the authorization prompt.
     */
    public function skipsAuthorization(): bool
    {
        return $this->firstParty();
    }
}
```

Converting Authorization Codes to Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should first verify the `state` parameter against the value that was stored prior to the redirect. If the state parameter matches then the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

Route::get('/callback', function (Request $request) {
    $state = $request->session()->pull('state');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class,
        'Invalid state value.'
    );

    $response = Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code' => $request->code,
    ]);

    return $response->json();
});
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.



Like the `/oauth/authorize` route, the `/oauth/token` route is defined for you by Passport. There is no need to manually define this route.

JSON API

Passport also includes a JSON API for managing authorized access tokens. You may pair this with your own frontend to offer your users a dashboard for managing access tokens. For convenience, we'll use [Axios](#) to demonstrate making HTTP requests to the endpoints. The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application.

GET `/oauth/tokens`

This route returns all of the authorized access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they can revoke them:

```
axios.get('/oauth/tokens')
  .then(response => {
    console.log(response.data);
  });
}
```

DELETE `/oauth/tokens/{token-id}`

This route may be used to revoke authorized access tokens and their related refresh tokens:

```
axios.delete('/oauth/tokens/' + tokenId);
```

Refreshing Tokens

If your application issues short-lived access tokens, users will need to refresh their access tokens via the refresh token that was provided to them when the access token was issued:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
  'grant_type' => 'refresh_token',
  'refresh_token' => 'the-refresh-token',
  'client_id' => 'client-id',
  'client_secret' => 'client-secret',
  'scope' => '',
]);

return $response->json();
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

Revoking Tokens

You may revoke a token by using the `revokeAccessToken` method on the `Laravel\Passport\TokenRepository`. You may revoke a token's refresh tokens using the `revokeRefreshTokensByAccessTokenId` method on the `Laravel\Passport\RefreshTokenRepository`. These classes may be resolved using Laravel's [service container](#):

```

use Laravel\Passport\TokenRepository;
use Laravel\Passport\RefreshTokenRepository;

$tokenRepository = app(TokenRepository::class);
$refreshTokenRepository = app(RefreshTokenRepository::class);

// Revoke an access token...
$tokenRepository->revokeAccessToken($tokenId);

// Revoke all of the token's refresh tokens...
$refreshTokenRepository->revokeRefreshTokensByAccessToken($tokenId);

```

Purging Tokens

When tokens have been revoked or expired, you might want to purge them from the database. Passport's included `passport:purge` Artisan command can do this for you:

```

# Purge revoked and expired tokens and auth codes...
php artisan passport:purge

# Only purge tokens expired for more than 6 hours...
php artisan passport:purge --hours=6

# Only purge revoked tokens and auth codes...
php artisan passport:purge --revoked

# Only purge expired tokens and auth codes...
php artisan passport:purge --expired

```

You may also configure a scheduled job in your application's `App\Console\Kernel` class to automatically prune your tokens on a schedule:

```

/**
 * Define the application's command schedule.
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('passport:purge')->hourly();
}

```

Authorization Code Grant With PKCE

The Authorization Code grant with "Proof Key for Code Exchange" (PKCE) is a secure way to authenticate single page applications or native applications to access your API. This grant should be used when you can't guarantee that the client secret will be stored confidentially or in order to mitigate the threat of having the authorization code intercepted by an attacker. A combination of a "code verifier" and a "code challenge" replaces the client secret when exchanging the authorization code for an access token.

Creating the Client

Before your application can issue tokens via the authorization code grant with PKCE, you will need to create a PKCE-enabled client. You may do this using the `passport:client` Artisan command with the `--public` option:

```
php artisan passport:client --public
```

Requesting Tokens

Code Verifier and Code Challenge

As this authorization grant does not provide a client secret, developers will need to generate a combination of a code verifier and a code challenge in order to request a token.

The code verifier should be a random string of between 43 and 128 characters containing letters, numbers, and `"-"`, `".."`, `"_"`, `"~"` characters, as defined in the [RFC 7636 specification](#).

The code challenge should be a Base64 encoded string with URL and filename-safe characters. The trailing `'='` characters should be removed and no line breaks, whitespace, or other additional characters should be present.

```
$encoded = base64_encode(hash('sha256', $code_verifier, true));

$codeChallenge = substr(rtrim($encoded, '='), '+/', '_');
```

Redirecting for Authorization

Once a client has been created, you may use the client ID and the generated code verifier and code challenge to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route:

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $request->session()->put(
        'code_verifier', $code_verifier = Str::random(128)
    );

    $codeChallenge = substr(rtrim(
        base64_encode(hash('sha256', $code_verifier, true))
        , '='), '+/', '_');

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        'code_challenge' => $codeChallenge,
        'code_challenge_method' => 'S256',
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?' . $query);
});
```

Converting Authorization Codes to Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should verify the `state` parameter against the value that was stored prior to the redirect, as in the standard Authorization Code Grant.

If the state parameter matches, the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request along with the originally generated code verifier:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

Route::get('/callback', function (Request $request) {
    $state = $request->session()->pull('state');

    $codeVerifier = $request->session()->pull('code_verifier');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response = Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code_verifier' => $codeVerifier,
        'code' => $request->code,
    ]);

    return $response->json();
});
```

Password Grant Tokens



We no longer recommend using password grant tokens. Instead, you should choose [a grant type that is currently recommended by OAuth2 Server](#).

The OAuth2 password grant allows your other first-party clients, such as a mobile application, to obtain an access token using an email address / username and password. This allows you to issue access tokens securely to your first-party clients without requiring your users to go through the entire OAuth2 authorization code redirect flow.

Creating a Password Grant Client

Before your application can issue tokens via the password grant, you will need to create a password grant client. You may do this using the `passport:client` Artisan command with the `--password` option. **If you have already run the `passport:install` command, you do not need to run this command:**

```
php artisan passport:client --password
```

Requesting Tokens

Once you have created a password grant client, you may request an access token by issuing a `POST` request to the `/oauth/token` route with the user's email address and password. Remember, this route is already registered by Passport so there is no need to define it manually. If the request is successful, you will receive an `access_token` and `refresh_token` in the JSON response from the server:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '',
]);

return $response->json();
```



Remember, access tokens are long-lived by default. However, you are free to [configure your maximum access token lifetime](#) if needed.

Requesting All Scopes

When using the password grant or client credentials grant, you may wish to authorize the token for all of the scopes supported by your application. You can do this by requesting the `*` scope. If you request the `*` scope, the `can` method on the token instance will always return `true`. This scope may only be assigned to a token that is issued using the `password` or `client_credentials` grant:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '*',
]);
```

Customizing the User Provider

If your application uses more than one `authentication user provider`, you may specify which user provider the password grant client uses by providing a `--provider` option when creating the client via the `artisan passport:client --password` command. The given provider name should match a valid provider defined in your application's `config/auth.php` configuration file. You can then `protect your route using middleware` to ensure that only users from the guard's specified provider are authorized.

Customizing the Username Field

When authenticating using the password grant, Passport will use the `email` attribute of your authenticatable model as the "username". However, you may customize this behavior by defining a `findForPassport` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Find the user instance for the given username.
     */
    public function findForPassport(string $username): User
    {
        return $this->where('username', $username)->first();
    }
}
```

Customizing the Password Validation

When authenticating using the password grant, Passport will use the `password` attribute of your model to validate the given password. If your model does not have a `password` attribute or you wish to customize the password validation logic, you can define a `validateForPassportPasswordGrant` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Hash;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Validate the password of the user for the Passport password grant.
     */
    public function validateForPassportPasswordGrant(string $password): bool
    {
        return Hash::check($password, $this->password);
    }
}
```

Implicit Grant Tokens



We no longer recommend using implicit grant tokens. Instead, you should choose [a grant type that is currently recommended by OAuth2 Server](#).

The implicit grant is similar to the authorization code grant; however, the token is returned to the client without exchanging an authorization code. This grant is most commonly used for JavaScript or mobile applications where the client credentials can't be securely stored. To enable the grant, call the `enableImplicitGrant` method in the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::enableImplicitGrant();
}
```

Once the grant has been enabled, developers may use their client ID to request an access token from your application. The consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
use Illuminate\Http\Request;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'token',
        'scope' => '',
        'state' => $state,
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```



Remember, the `/oauth/authorize` route is already defined by Passport. You do not need to manually define this route.

Client Credentials Grant Tokens

The client credentials grant is suitable for machine-to-machine authentication. For example, you might use this grant in a scheduled job which is performing maintenance tasks over an API.

Before your application can issue tokens via the client credentials grant, you will need to create a client credentials grant client. You may do this using the `--client` option of the `passport:client` Artisan command:

```
php artisan passport:client --client
```

Next, to use this grant type, you may add the `CheckClientCredentials` middleware to the `$middlewareAliases` property of your application's `app/Http/Kernel.php` file:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $middlewareAliases = [
    'client' => CheckClientCredentials::class,
];
```

Then, attach the middleware to a route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client');
```

To restrict access to the route to specific scopes, you may provide a comma-delimited list of the required scopes when attaching the `client` middleware to the route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client:check-status,your-scope');
```

Retrieving Tokens

To retrieve a token using this grant type, make a request to the `oauth/token` endpoint:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'client_credentials',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => 'your-scope',
]);

return $response->json()['access_token'];
```

Personal Access Tokens

Sometimes, your users may want to issue access tokens to themselves without going through the typical authorization code redirect flow. Allowing users to issue tokens to themselves via your application's UI can be useful for allowing users to experiment with your API or may serve as a simpler approach to issuing access tokens in general.



If your application is primarily using Passport to issue personal access tokens, consider using [Laravel Sanctum](#), Laravel's light-weight first-party library for issuing API access tokens.

Creating a Personal Access Client

Before your application can issue personal access tokens, you will need to create a personal access client. You may do this by executing the `passport:client` Artisan command with the `--personal` option. If you have already run the `passport:install` command, you do not need to run this command:

```
php artisan passport:client --personal
```

After creating your personal access client, place the client's ID and plain-text secret value in your application's `.env` file:

```
PASSPORT_PERSONAL_ACCESS_CLIENT_ID="client-id-value"
PASSPORT_PERSONAL_ACCESS_CLIENT_SECRET="unhashed-client-secret-value"
```

Managing Personal Access Tokens

Once you have created a personal access client, you may issue tokens for a given user using the `createToken` method on the `App\Models\User` model instance. The `createToken` method accepts the name of the token as its first argument and an optional array of scopes as its second argument:

```
use App\Models\User;

$user = User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport also includes a JSON API for managing personal access tokens. You may pair this with your own frontend to offer your users a dashboard for managing personal access tokens. Below, we'll review all of the API endpoints for managing personal access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

GET /oauth/scopes

This route returns all of the scopes defined for your application. You may use this route to list the scopes a user may assign to a personal access token:

```
axios.get('/oauth/scopes')
  .then(response => {
    console.log(response.data);
  });
```

GET /oauth/personal-access-tokens

This route returns all of the personal access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they may edit or revoke them:

```
axios.get('/oauth/personal-access-tokens')
  .then(response => {
    console.log(response.data);
  });
}
```

POST /oauth/personal-access-tokens

This route creates new personal access tokens. It requires two pieces of data: the token's `name` and the `scopes` that should be assigned to the token:

```
const data = {
  name: 'Token Name',
  scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
  .then(response => {
    console.log(response.data.accessToken);
  })
  .catch(response => {
    // List errors on response...
  });
}
```

DELETE /oauth/personal-access-tokens/{token-id}

This route may be used to revoke personal access tokens:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protecting Routes

Via Middleware

Passport includes an `authentication_guard` that will validate access tokens on incoming requests. Once you have configured the `api` guard to use the `passport` driver, you only need to specify the `auth:api` middleware on any routes that should require a valid access token:

```
Route::get('/user', function () {
  // ...
})->middleware('auth:api');
```



If you are using the `client credentials grant`, you should use the `client` middleware to protect your routes instead of the `auth:api` middleware.

Multiple Authentication Guards

If your application authenticates different types of users that perhaps use entirely different Eloquent models, you will likely need to define a guard configuration for each user provider type in your application. This allows you to protect requests intended for specific user providers. For example, given the following guard configuration the `config/auth.php` configuration file:

```
'api' => [
    'driver' => 'passport',
    'provider' => 'users',
],

'api-customers' => [
    'driver' => 'passport',
    'provider' => 'customers',
],
```

The following route will utilize the `api-customers` guard, which uses the `customers` user provider, to authenticate incoming requests:

```
Route::get('/customer', function () {
    // ...
})->middleware('auth:api-customers');
```



For more information on using multiple user providers with Passport, please consult the [password grant documentation](#).

Passing the Access Token

When calling routes that are protected by Passport, your application's API consumers should specify their access token as a `Bearer` token in the `Authorization` header of their request. For example, when using the Guzzle HTTP library:

```
use Illuminate\Support\Facades\Http;

$response = Http::withHeaders([
    'Accept' => 'application/json',
    'Authorization' => 'Bearer ' . $accessToken,
])->get('https://passport-app.test/api/user');

return $response->json();
```

Token Scopes

Scopes allow your API clients to request a specific set of permissions when requesting authorization to access an account. For example, if you are building an e-commerce application, not all API consumers will need the ability to place orders. Instead, you may allow the consumers to only request authorization to access order shipment statuses. In other words, scopes allow your application's users to limit the actions a third-party application can perform on their behalf.

Defining Scopes

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your application's `App\Providers\AuthServiceProvider` class. The `tokensCan` method accepts an array of scope names and scope descriptions. The scope description may be anything you wish and will be displayed to users on the authorization approval screen:

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::tokensCan([
        'place-orders' => 'Place orders',
        'check-status' => 'Check order status',
    ]);
}
```

Default Scope

If a client does not request any specific scopes, you may configure your Passport server to attach default scope(s) to the token using the `setDefaultScope` method. Typically, you should call this method from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);

Passport::setDefaultScope([
    'check-status',
    'place-orders',
]);
```



Passport's default scopes do not apply to personal access tokens that are generated by the user.

Assigning Scopes to Tokens

When Requesting Authorization Codes

When requesting an access token using the authorization code grant, consumers should specify their desired scopes as the `scope` query string parameter. The `scope` parameter should be a space-delimited list of scopes:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `App\Models\User` model's `createToken` method, you may pass the array of desired scopes as the second argument to the method:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Checking Scopes

Passport includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given scope. To get started, add the following middleware to the `$middlewareAliases` property of your `app/Http/Kernel.php` file:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Check For All Scopes

The `scopes` middleware may be assigned to a route to verify that the incoming request's access token has all of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
})->middleware(['auth:api', 'scopes:check-status,place-orders']);
```

Check for Any Scopes

The `scope` middleware may be assigned to a route to verify that the incoming request's access token has *at least one* of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
})->middleware(['auth:api', 'scope:check-status,place-orders']);
```

Checking Scopes on a Token Instance

Once an access token authenticated request has entered your application, you may still check if the token has a given scope using the `tokenCan` method on the authenticated `App\Models\User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        // ...
    }
});
```

Additional Scope Methods

The `scopeIds` method will return an array of all defined IDs / names:

```
use Laravel\Passport\Passport;

Passport::scopeIds();
```

The `scopes` method will return an array of all defined scopes as instances of `Laravel\Passport\Scope`:

```
Passport::scopes();
```

The `scopesFor` method will return an array of `Laravel\Passport\Scope` instances matching the given IDs / names:

```
Passport::scopesFor(['place-orders', 'check-status']);
```

You may determine if a given scope has been defined using the `hasScope` method:

```
Passport::hasScope('place-orders');
```

Consuming Your API With JavaScript

When building an API, it can be extremely useful to be able to consume your own API from your JavaScript application. This approach to API development allows your own application to consume the same API that you are sharing with the world.

The same API may be consumed by your web application, mobile applications, third-party applications, and any SDKs that you may publish on various package managers.

Typically, if you want to consume your API from your JavaScript application, you would need to manually send an access token to the application and pass it with each request to your application. However, Passport includes a middleware that can handle this for you. All you need to do is add the `CreateFreshApiToken` middleware to your `web` middleware group in your `app/Http/Kernel.php` file:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```



You should ensure that the `CreateFreshApiToken` middleware is the last middleware listed in your middleware stack.

This middleware will attach a `laravel_token` cookie to your outgoing responses. This cookie contains an encrypted JWT that Passport will use to authenticate API requests from your JavaScript application. The JWT has a lifetime equal to your `session.lifetime` configuration value. Now, since the browser will automatically send the cookie with all subsequent requests, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/api/user')
    .then(response => {
        console.log(response.data);
   });
```

Customizing the Cookie Name

If needed, you can customize the `laravel_token` cookie's name using the `Passport::cookie` method. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::cookie('custom_name');
}
```

CSRF Protection

When using this method of authentication, you will need to ensure a valid CSRF token header is included in your requests. The default Laravel JavaScript scaffolding includes an Axios instance, which will automatically use the encrypted `XSRF-TOKEN` cookie value to send an `X-XSRF-TOKEN` header on same-origin requests.



If you choose to send the `X-CSRF-TOKEN` header instead of `X-XSRF-TOKEN`, you will need to use the unencrypted token provided by `csrf_token()`.

Events

Passport raises events when issuing access tokens and refresh tokens. You may use these events to prune or revoke other access tokens in your database. If you would like, you may attach listeners to these events in your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],
    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

Testing

Passport's `actingAs` method may be used to specify the currently authenticated user as well as its scopes. The first argument given to the `actingAs` method is the user instance and the second is an array of scopes that should be granted to the user's token:

```
use App\Models\User;
use Laravel\Passport\Passport;

public function test_servers_can_be_created(): void
{
    Passport::actingAs(
        User::factory()->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(201);
}
```

Passport's `actingAsClient` method may be used to specify the currently authenticated client as well as its scopes. The first argument given to the `actingAsClient` method is the client instance and the second is an array of scopes that should be granted to the client's token:

```
use Laravel\Passport\Client;
use Laravel\Passport\Passport;

public function test_orders_can_be_retrieved(): void
{
    Passport::actingAsClient(
        Client::factory()->create(),
        ['check-status']
    );

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

    $response->assertStatus(200);
}
```

Laravel Pennant

- [Introduction](#)
- [Installation](#)
- [Configuration](#)
- [Defining Features](#)
 - [Class Based Features](#)
- [Checking Features](#)
 - [Conditional Execution](#)
 - [The `HasFeatures` Trait](#)
 - [Blade Directive](#)
 - [Middleware](#)
 - [In-Memory Cache](#)
- [Scope](#)
 - [Specifying the Scope](#)
 - [Default Scope](#)
 - [Nullable Scope](#)
 - [Identifying Scope](#)
 - [Serializing Scope](#)
- [Rich Feature Values](#)
- [Retrieving Multiple Features](#)
- [Eager Loading](#)
- [Updating Values](#)
 - [Bulk Updates](#)
 - [Purging Features](#)
- [Testing](#)
- [Adding Custom Pennant Drivers](#)
 - [Implementing the Driver](#)
 - [Registering the Driver](#)
- [Events](#)

Introduction

[Laravel Pennant](#) is a simple and light-weight feature flag package - without the cruft. Feature flags enable you to incrementally roll out new application features with confidence, A/B test new interface designs, complement a trunk-based development strategy, and much more.

Installation

First, install Pennant into your project using the Composer package manager:

```
composer require laravel/pennant
```

Next, you should publish the Pennant configuration and migration files using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --provider="Laravel\Pennant\PennantServiceProvider"
```

Finally, you should run your application's database migrations. This will create a `features` table that Pennant uses to power its `database` driver:

```
php artisan migrate
```

Configuration

After publishing Pennant's assets, its configuration file will be located at `config/pennant.php`. This configuration file allows you to specify the default storage mechanism that will be used by Pennant to store resolved feature flag values.

Pennant includes support for storing resolved feature flag values in an in-memory array via the `array` driver. Or, Pennant can store resolved feature flag values persistently in a relational database via the `database` driver, which is the default storage mechanism used by Pennant.

Defining Features

To define a feature, you may use the `define` method offered by the `Feature` facade. You will need to provide a name for the feature, as well as a closure that will be invoked to resolve the feature's initial value.

Typically, features are defined in a service provider using the `Feature` facade. The closure will receive the "scope" for the feature check. Most commonly, the scope is the currently authenticated user. In this example, we will define a feature for incrementally rolling out a new API to our application's users:

```
<?php

namespace App\Providers;

use App\Models\User;
use Illuminate\Support\Lottery;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Feature::define('new-api', fn (User $user) => match ($true) {
            $user->isInternalTeamMember() => true,
            $user->isHighTrafficCustomer() => false,
            default => Lottery::odds(1 / 100),
        });
    }
}
```

As you can see, we have the following rules for our feature:

- All internal team members should be using the new API.
- Any high traffic customers should not be using the new API.
- Otherwise, the feature should be randomly assigned to users with a 1 in 100 chance of being active.

The first time the `new-api` feature is checked for a given user, the result of the closure will be stored by the storage driver. The next time the feature is checked against the same user, the value will be retrieved from storage and the closure will not be invoked.

For convenience, if a feature definition only returns a lottery, you may omit the closure completely:

```
Feature::define('site-redesign', Lottery::odds(1, 1000));
```

Class Based Features

Pennant also allows you to define class based features. Unlike closure based feature definitions, there is no need to register a class based feature in a service provider. To create a class based feature, you may invoke the `pennant:feature` Artisan command. By default the feature class will be placed in your application's `app/Features`

directory:

```
php artisan pennant:feature NewApi
```

When writing a feature class, you only need to define a `resolve` method, which will be invoked to resolve the feature's initial value for a given scope. Again, the scope will typically be the currently authenticated user:

```
<?php

namespace App\Features;

use Illuminate\Support\Lottery;

class NewApi
{
    /**
     * Resolve the feature's initial value.
     */
    public function resolve(User $user): mixed
    {
        return match (true) {
            $user->isInternalTeamMember() => true,
            $user->isHighTrafficCustomer() => false,
            default => Lottery::odds(1 / 100),
        };
    }
}
```



Feature classes are resolved via the [container](#), so you may inject dependencies into the feature class's constructor when needed.

Customizing the Stored Feature Name

By default, Pennant will store the feature class's fully qualified class name. If you would like to decouple the stored feature name from the application's internal structure, you may specify a `$name` property on the feature class. The value of this property will be stored in place of the class name:

```
<?php

namespace App\Features;

class NewApi
{
    /**
     * The stored name of the feature.
     *
     * @var string
     */
    public $name = 'new-api';

    // ...
}
```

Checking Features

To determine if a feature is active, you may use the `active` method on the `Feature` facade. By default, features are checked against the currently authenticated user:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     * Display a listing of the resource.
     */
    public function index(Request $request): Response
    {
        return Feature::active('new-api')
            ? $this->resolveNewApiResponse($request)
            : $this->resolveLegacyApiResponse($request);
    }

    // ...
}
```

Although features are checked against the currently authenticated user by default, you may easily check the feature against another user or `scope`. To accomplish this, use the `for` method offered by the `Feature` facade:

```
return Feature::for($user)->active('new-api')
    ? $this->resolveNewApiResponse($request)
    : $this->resolveLegacyApiResponse($request);
```

Pennant also offers some additional convenience methods that may prove useful when determining if a feature is active or not:

```
// Determine if all of the given features are active...
Feature::allAreActive(['new-api', 'site-redesign']);

// Determine if any of the given features are active...
Feature::someAreActive(['new-api', 'site-redesign']);

// Determine if a feature is inactive...
Feature::inactive('new-api');

// Determine if all of the given features are inactive...
Feature::allAreInactive(['new-api', 'site-redesign']);

// Determine if any of the given features are inactive...
Feature::someAreInactive(['new-api', 'site-redesign']);
```



When using Pennant outside of an HTTP context, such as in an Artisan command or a queued job, you should typically explicitly specify the feature's scope. Alternatively, you may define a default scope that accounts for both authenticated HTTP contexts and unauthenticated contexts.

Checking Class Based Features

For class based features, you should provide the class name when checking the feature:

```
<?php

namespace App\Http\Controllers;

use App\Features\NewApi;
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     * Display a listing of the resource.
     */
    public function index(Request $request): Response
    {
        return Feature::active(NewApi::class)
            ? $this->resolveNewApiResponse($request)
            : $this->resolveLegacyApiResponse($request);
    }

    // ...
}
```

Conditional Execution

The `when` method may be used to fluently execute a given closure if a feature is active. Additionally, a second closure may be provided and will be executed if the feature is inactive:

```
<?php

namespace App\Http\Controllers;

use App\Features\NewApi;
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     * Display a listing of the resource.
     */
    public function index(Request $request): Response
    {
        return Feature::when(NewApi::class,
            fn () => $this->resolveNewApiResponse($request),
            fn () => $this->resolveLegacyApiResponse($request),
        );
    }

    // ...
}
```

The `unless` method serves as the inverse of the `when` method, executing the first closure if the feature is inactive:

```
return Feature::unless(NewApi::class,
    fn () => $this->resolveLegacyApiResponse($request),
    fn () => $this->resolveNewApiResponse($request),
);
```

The `HasFeatures` Trait

Pennant's `HasFeatures` trait may be added to your application's `User` model (or any other model that has features) to provide a fluent, convenient way to check features directly from the model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Laravel\Pennant\Concerns\HasFeatures;

class User extends Authenticatable
{
    use HasFeatures;

    // ...
}
```

Once the trait has been added to your model, you may easily check features by invoking the `features` method:

```
if ($user->features()->active('new-api')) {
    // ...
}
```

Of course, the `features` method provides access to many other convenient methods for interacting with features:

```
// Values...
$value = $user->features()->value('purchase-button')
$values = $user->features()->values(['new-api', 'purchase-button']);

// State...
$user->features()->active('new-api');
$user->features()->allAreActive(['new-api', 'server-api']);
$user->features()->someAreActive(['new-api', 'server-api']);

$user->features()->inactive('new-api');
$user->features()->allAreInactive(['new-api', 'server-api']);
$user->features()->someAreInactive(['new-api', 'server-api']);

// Conditional execution...
$user->features()->when('new-api',
    fn () => /* ... */,
    fn () => /* ... */,
);
$user->features()->unless('new-api',
    fn () => /* ... */,
    fn () => /* ... */,
);
```

Blade Directive

To make checking features in Blade a seamless experience, Pennant offers a `@feature` directive:

```
@feature('site-redesign')
    <!-- 'site-redesign' is active -->
@else
    <!-- 'site-redesign' is inactive -->
@endfeature
```

Middleware

Pennant also includes a `middleware` that may be used to verify the currently authenticated user has access to a feature before a route is even invoked. You may assign the middleware to a route and specify the features that are required to access the route. If any of the specified features are inactive for the currently authenticated user, a `400 Bad Request` HTTP response will be returned by the route. Multiple features may be passed to the static `using` method.

```
use Illuminate\Support\Facades\Route;
use Laravel\Pennant\Middleware\EnsureFeaturesAreActive;

Route::get('/api/servers', function () {
    // ...
})->middleware(EnsureFeaturesAreActive::using('new-api', 'servers-api'));
```

Customizing the Response

If you would like to customize the response that is returned by the middleware when one of the listed features is inactive, you may use the `whenInactive` method provided by the `EnsureFeaturesAreActive` middleware. Typically, this method should be invoked within the `boot` method of one of your application's service providers:

```
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use LaravelPennant\Middleware\EnsureFeaturesAreActive;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    EnsureFeaturesAreActive::whenInactive(
        function (Request $request, array $features) {
            return new Response(status: 403);
        }
    );

    // ...
}
```

In-Memory Cache

When checking a feature, Pennant will create an in-memory cache of the result. If you are using the `database` driver, this means that re-checking the same feature flag within a single request will not trigger additional database queries. This also ensures that the feature has a consistent result for the duration of the request.

If you need to manually flush the in-memory cache, you may use the `flushCache` method offered by the `Feature` facade:

```
Feature::flushCache();
```

Scope

Specifying the Scope

As discussed, features are typically checked against the currently authenticated user. However, this may not always suit your needs. Therefore, it is possible to specify the scope you would like to check a given feature against via the `Feature` facade's `for` method:

```
return Feature::for($user)->active('new-api')
    ? $this->resolveNewApiResponse($request)
    : $this->resolveLegacyApiResponse($request);
```

Of course, feature scopes are not limited to "users". Imagine you have built a new billing experience that you are rolling out to entire teams rather than individual users. Perhaps you would like the oldest teams to have a slower rollout than the newer teams. Your feature resolution closure might look something like the following:

```

use App\Models\Team;
use Carbon\Carbon;
use Illuminate\Support\Lottery;
use Laravel\Pennant\Feature;

Feature::define('billing-v2', function (Team $team) {
    if ($team->created_at->isAfter(new Carbon('1st Jan, 2023'))) {
        return true;
    }

    if ($team->created_at->isAfter(new Carbon('1st Jan, 2019'))) {
        return Lottery::odds(1 / 100);
    }

    return Lottery::odds(1 / 1000);
});

```

You will notice that the closure we have defined is not expecting a `User`, but is instead expecting a `Team` model. To determine if this feature is active for a user's team, you should pass the team to the `for` method offered by the `Feature` facade:

```

if (Feature::for($user->team)->active('billing-v2')) {
    return redirect()->to('/billing/v2');
}

// ...

```

Default Scope

It is also possible to customize the default scope Pennant uses to check features. For example, maybe all of your features are checked against the currently authenticated user's team instead of the user. Instead of having to call `Feature::for($user->team)` every time you check a feature, you may instead specify the team as the default scope. Typically, this should be done in one of your application's service providers:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Feature::resolveScopeUsing(fn ($driver) => Auth::user()?->team);

        // ...
    }
}

```

If no scope is explicitly provided via the `for` method, the feature check will now use the currently authenticated user's team as the default scope:

```
Feature::active('billing-v2');

// Is now equivalent to...

Feature::for($user->team)->active('billing-v2');
```

Nullable Scope

If the scope you provide when checking a feature is `null` and the feature's definition does not support `null` via a nullable type or by including `null` in a union type, Pennant will automatically return `false` as the feature's result value.

So, if the scope you are passing to a feature is potentially `null` and you want the feature's value resolver to be invoked, you should account for that in your feature's definition. A `null` scope may occur if you check a feature within an Artisan command, queued job, or unauthenticated route. Since there is usually not an authenticated user in these contexts, the default scope will be `null`.

If you do not always explicitly specify your feature scope then you should ensure the scope's type is "nullable" and handle the `null` scope value within your feature definition logic:

```
use App\Models\User;
use Illuminate\Support\Lottery;
use Laravel\Pennant\Feature;

- Feature::define('new-api', fn (User $user) => match (true) {
+ Feature::define('new-api', fn (User|null $user) => match (true) {
+     $user === null => true,
         $user->isInternalTeamMember() => true,
         $user->isHighTrafficCustomer() => false,
         default => Lottery::odds(1 / 100),
    );
}
```

Identifying Scope

Pennant's built-in `array` and `database` storage drivers know how to properly store scope identifiers for all PHP data types as well as Eloquent models. However, if your application utilizes a third-party Pennant driver, that driver may not know how to properly store an identifier for an Eloquent model or other custom types in your application.

In light of this, Pennant allows you to format scope values for storage by implementing the `FeatureScopeable` contract on the objects in your application that are used as Pennant scopes.

For example, imagine you are using two different feature drivers in a single application: the built-in `database` driver and a third-party "Flag Rocket" driver. The "Flag Rocket" driver does not know how to properly store an Eloquent model. Instead, it requires a `FlagRocketUser` instance. By implementing the `toFeatureIdentifier` defined by the `FeatureScopeable` contract, we can customize the storablescope value provided to each driver used by our application:

```
<?php

namespace App\Models;

use FlagRocket\FlagRocketUser;
use Illuminate\Database\Eloquent\Model;
use Laravel\Pennant\Contracts\FeatureScopeable;

class User extends Model implements FeatureScopeable
{
    /**
     * Cast the object to a feature scope identifier for the given driver.
     */
    public function toFeatureIdentifier(string $driver): mixed
    {
        return match($driver) {
            'database' => $this,
            'flag-rocket' => FlagRocketUser::fromId($this->flag_rocket_id),
        };
    }
}
```

Serializing Scope

By default, Pennant will use a fully qualified class name when storing a feature associated with an Eloquent model. If you are already using an [Eloquent morph map](#), you may choose to have Pennant also use the morph map to decouple the stored feature from your application structure.

To achieve this, after defining your Eloquent morph map in a service provider, you may invoke the `Feature` facade's `useMorphMap` method:

```
use Illuminate\Database\Eloquent\Relations\Relation;
use Laravel\Pennant\Feature;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);

Feature::useMorphMap();
```

Rich Feature Values

Until now, we have primarily shown features as being in a binary state, meaning they are either "active" or "inactive", but Pennant also allows you to store rich values as well.

For example, imagine you are testing three new colors for the "Buy now" button of your application. Instead of returning `true` or `false` from the feature definition, you may instead return a string:

```
use Illuminate\Support\Arr;
use Laravel\Pennant\Feature;

Feature::define('purchase-button', fn (User $user) => Arr::random([
    'blue-sapphire',
    'seafoam-green',
    'tart-orange',
]));

```

You may retrieve the value of the `purchase-button` feature using the `value` method:

```
$color = Feature::value('purchase-button');
```

Pennant's included Blade directive also makes it easy to conditionally render content based on the current value of the feature:

```
@feature('purchase-button', 'blue-sapphire')
    <!-- 'blue-sapphire' is active -->
@elseifeature('purchase-button', 'seafoam-green')
    <!-- 'seafoam-green' is active -->
@elseifeature('purchase-button', 'tart-orange')
    <!-- 'tart-orange' is active -->
@endfeature
```



When using rich values, it is important to know that a feature is considered "active" when it has any value other than `false`.

When calling the `conditional when` method, the feature's rich value will be provided to the first closure:

```
Feature::when('purchase-button',
    fn ($color) => /* ... */,
    fn () => /* ... */,
);
```

Likewise, when calling the `conditional unless` method, the feature's rich value will be provided to the optional second closure:

```
Feature::unless('purchase-button',
    fn () => /* ... */,
    fn ($color) => /* ... */,
);
```

Retrieving Multiple Features

The `values` method allows the retrieval of multiple features for a given scope:

```
Feature::values(['billing-v2', 'purchase-button']);

// [
//     'billing-v2' => false,
//     'purchase-button' => 'blue-sapphire',
// ]
```

Or, you may use the `all` method to retrieve the values of all defined features for a given scope:

```
Feature::all();

// [
//     'billing-v2' => false,
//     'purchase-button' => 'blue-sapphire',
//     'site-redesign' => true,
// ]
```

However, class based features are dynamically registered and are not known by Pennant until they are explicitly checked. This means your application's class based features may not appear in the results returned by the `all` method if they have not already been checked during the current request.

If you would like to ensure that feature classes are always included when using the `all` method, you may use Pennant's feature discovery capabilities. To get started, invoke the `discover` method in one of your application's service providers:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Feature::discover();

        // ...
    }
}
```

The `discover` method will register all of the feature classes in your application's `app/Features` directory. The `all` method will now include these classes in its results, regardless of whether they have been checked during the current request:

```
Feature::all();

// [
//     'App\Features\NewApi' => true,
//     'billing-v2' => false,
//     'purchase-button' => 'blue-sapphire',
//     'site-redesign' => true,
// ]
```

Eager Loading

Although Pennant keeps an in-memory cache of all resolved features for a single request, it is still possible to encounter performance issues. To alleviate this, Pennant offers the ability to eager load feature values.

To illustrate this, imagine that we are checking if a feature is active within a loop:

```
use Laravel\Pennant\Feature;

foreach ($users as $user) {
    if (Feature::for($user)->active('notifications-beta')) {
        $user->notify(new RegistrationSuccess);
    }
}
```

Assuming we are using the database driver, this code will execute a database query for every user in the loop - executing potentially hundreds of queries. However, using Pennant's `load` method, we can remove this potential performance bottleneck by eager loading the feature values for a collection of users or scopes:

```
Feature::for($users)->load(['notifications-beta']);

foreach ($users as $user) {
    if (Feature::for($user)->active('notifications-beta')) {
        $user->notify(new RegistrationSuccess);
    }
}
```

To load feature values only when they have not already been loaded, you may use the `loadMissing` method:

```
Feature::for($users)->loadMissing([
    'new-api',
    'purchase-button',
    'notifications-beta',
]);
```

Updating Values

When a feature's value is resolved for the first time, the underlying driver will store the result in storage. This is often necessary to ensure a consistent experience for your users across requests. However, at times, you may want to manually update the feature's stored value.

To accomplish this, you may use the `activate` and `deactivate` methods to toggle a feature "on" or "off":

```
use Laravel\Pennant\Feature;

// Activate the feature for the default scope...
Feature::activate('new-api');

// Deactivate the feature for the given scope...
Feature::for($user->team)->deactivate('billing-v2');
```

It is also possible to manually set a rich value for a feature by providing a second argument to the `activate` method:

```
Feature::activate('purchase-button', 'seafoam-green');
```

To instruct Pennant to forget the stored value for a feature, you may use the `forget` method. When the feature is checked again, Pennant will resolve the feature's value from its feature definition:

```
Feature::forget('purchase-button');
```

Bulk Updates

To update stored feature values in bulk, you may use the `activateForEveryone` and `deactivateForEveryone` methods.

For example, imagine you are now confident in the `'new-api'` feature's stability and have landed on the best `'purchase-button'` color for your checkout flow - you can update the stored value for all users accordingly:

```
use Laravel\Pennant\Feature;

Feature::activateForEveryone('new-api');

Feature::activateForEveryone('purchase-button', 'seafoam-green');
```

Alternatively, you may deactivate the feature for all users:

```
Feature::deactivateForEveryone('new-api');
```



This will only update the resolved feature values that have been stored by Pennant's storage driver. You will also need to update the feature definition in your application.

Purging Features

Sometimes, it can be useful to purge an entire feature from storage. This is typically necessary if you have removed the feature from your application or you have made adjustments to the feature's definition that you would like to rollout to all users.

You may remove all stored values for a feature using the `purge` method:

```
// Purging a single feature...
Feature::purge('new-api');

// Purging multiple features...
Feature::purge(['new-api', 'purchase-button']);
```

If you would like to purge *all* features from storage, you may invoke the `purge` method without any arguments:

```
Feature::purge();
```

As it can be useful to purge features as part of your application's deployment pipeline, Pennant includes a `pennant:purge` Artisan command which will purge the provided features from storage:

```
php artisan pennant:purge new-api

php artisan pennant:purge new-api purchase-button
```

It is also possible to purge all features *except* those in a given feature list. For example, imagine you wanted to purge all features but keep the values for the "new-api" and "purchase-button" features in storage. To accomplish this, you can pass those feature names to the `--except` option:

```
php artisan pennant:purge --except=new-api --except=purchase-button
```

For convenience, the `pennant:purge` command also supports an `--except-registered` flag. This flag indicates that all features except those explicitly registered in a service provider should be purged:

```
php artisan pennant:purge --except-registered
```

Testing

When testing code that interacts with feature flags, the easiest way to control the feature flag's returned value in your tests is to simply re-define the feature. For example, imagine you have the following feature defined in one of your application's service provider:

```
use Illuminate\Support\Arr;
use Laravel\Pennant\Feature;

Feature::define('purchase-button', fn () => Arr::random([
    'blue-sapphire',
    'seafoam-green',
    'tart-orange',
]));
```

To modify the feature's returned value in your tests, you may re-define the feature at the beginning of the test. The following test will always pass, even though the `Arr::random()` implementation is still present in the service provider:

```
use Laravel\Pennant\Feature;

public function test_it_can_control_feature_values()
{
    Feature::define('purchase-button', 'seafoam-green');

    $this->assertSame('seafoam-green', Feature::value('purchase-button'));
}
```

The same approach may be used for class based features:

```
use App\Features\NewApi;
use Laravel\Pennant\Feature;

public function test_it_can_control_feature_values()
{
    Feature::define(NewApi::class, true);

    $this->assertTrue(Feature::value(NewApi::class));
}
```

If your feature is returning a `Lottery` instance, there are a handful of useful [testing helpers available](#).

Store Configuration

You may configure the store that Pennant will use during testing by defining the `PENNANT_STORE` environment variable in your application's `phpunit.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit colors="true">
    <!-- ... -->
    <php>
        <env name="PENNANT_STORE" value="array"/>
    <!-- ... -->
    </php>
</phpunit>
```

Adding Custom Pennant Drivers

Implementing the Driver

If none of Pennant's existing storage drivers fit your application's needs, you may write your own storage driver. Your custom driver should implement the `Laravel\Pennant\Contracts\Driver` interface:

```
<?php

namespace App\Extensions;

use Laravel\Pennant\Contracts\Driver;

class RedisFeatureDriver implements Driver
{
    public function define(string $feature, callable $resolver): void {}
    public function defined(): array {}
    public function getAll(array $features): array {}
    public function get(string $feature, mixed $scope): mixed {}
    public function set(string $feature, mixed $scope, mixed $value): void {}
    public function setForAllScopes(string $feature, mixed $value): void {}
    public function delete(string $feature, mixed $scope): void {}
    public function purge(array|null $features): void {}
}
```

Now, we just need to implement each of these methods using a Redis connection. For an example of how to implement each of these methods, take a look at the [Laravel\Pennant\Drivers\DatabaseDriver](#) in the [Pennant source code](#).



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 `RedisFeatureDriver`.

Registering the Driver

Once your driver has been implemented, you are ready to register it with Laravel. To add additional drivers to Pennant, you may use the `extend` method provided by the `Feature` facade. You should call the `extend` method from the `boot` method of one of your application's [service provider](#):

```
<?php

namespace App\Providers;

use App\Extensions\RedisFeatureDriver;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Feature::extend('redis', function (Application $app) {
            return new RedisFeatureDriver($app->make('redis'), $app->make('events'), []);
        });
    }
}
```

Once the driver has been registered, you may use the `redis` driver in your application's `config/pennant.php` configuration file:

```
'stores' => [
    'redis' => [
        'driver' => 'redis',
        'connection' => null,
    ],
    // ...
],
```

Events

Pennant dispatches a variety of events that can be useful when tracking feature flags throughout your application.

`Laravel\Pennant\Events\RetrievingKnownFeature`

This event is dispatched the first time a known feature is retrieved during a request for a specific scope. This event can be useful to create and track metrics against the feature flags that are being used throughout your application.

`Laravel\Pennant\Events\RetrievingUnknownFeature`

This event is dispatched the first time an unknown feature is retrieved during a request for a specific scope. This event can be useful if you have intended to remove a feature flag, but may have accidentally left some stray references to it throughout your application.

For example, you may find it useful to listen for this event and `report` or throw an exception when it occurs:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Event;
use Laravel\Pennant\Events\RetrievingUnknownFeature;

class EventServiceProvider extends ServiceProvider
{
    /**
     * Register any other events for your application.
     */
    public function boot(): void
    {
        Event::listen(function (RetrievingUnknownFeature $event) {
            report("Resolving unknown feature [{$event->feature}].");
        });
    }
}
```

Laravel\Pennant\Events\DynamicallyDefiningFeature

This event is dispatched when a class based feature is being dynamically checked for the first time during a request.

Laravel Pint

- [Introduction](#)
- [Installation](#)
- [Running Pint](#)
- [Configuring Pint](#)
 - [Presets](#)
 - [Rules](#)
 - [Excluding Files / Folders](#)

Introduction

[Laravel Pint](#) is an opinionated PHP code style fixer for minimalists. Pint is built on top of PHP-CS-Fixer and makes it simple to ensure that your code style stays clean and consistent.

Pint is automatically installed with all new Laravel applications so you may start using it immediately. By default, Pint does not require any configuration and will fix code style issues in your code by following the opinionated coding style of Laravel.

Installation

Pint is included in recent releases of the Laravel framework, so installation is typically unnecessary. However, for older applications, you may install Laravel Pint via Composer:

```
composer require laravel/pint --dev
```

Running Pint

You can instruct Pint to fix code style issues by invoking the `pint` binary that is available in your project's `vendor/bin` directory:

```
./vendor/bin/pint
```

You may also run Pint on specific files or directories:

```
./vendor/bin/pint app/Models
./vendor/bin/pint app/Models/User.php
```

Pint will display a thorough list of all of the files that it updates. You can view even more detail about Pint's changes by providing the `-v` option when invoking Pint:

```
./vendor/bin/pint -v
```

If you would like Pint to simply inspect your code for style errors without actually changing the files, you may use the `--test` option:

```
./vendor/bin/pint --test
```

If you would like Pint to only modify the files that have uncommitted changes according to Git, you may use the `--dirty` option:

```
./vendor/bin/pint --dirty
```

Configuring Pint

As previously mentioned, Pint does not require any configuration. However, if you wish to customize the presets, rules, or inspected folders, you may do so by creating a `pint.json` file in your project's root directory:

```
{
  "preset": "laravel"
}
```

In addition, if you wish to use a `pint.json` from a specific directory, you may provide the `--config` option when invoking Pint:

```
pint --config vendor/my-company/coding-style/pint.json
```

Presets

Presets defines a set of rules that can be used to fix code style issues in your code. By default, Pint uses the `laravel` preset, which fixes issues by following the opinionated coding style of Laravel. However, you may specify a different preset by providing the `--preset` option to Pint:

```
pint --preset psr12
```

If you wish, you may also set the preset in your project's `pint.json` file:

```
{
  "preset": "psr12"
}
```

Pint's currently supported presets are: `laravel`, `per`, `psr12`, and `symfony`.

Rules

Rules are style guidelines that Pint will use to fix code style issues in your code. As mentioned above, presets are predefined groups of rules that should be perfect for most PHP projects, so you typically will not need to worry about the individual rules they contain.

However, if you wish, you may enable or disable specific rules in your `pint.json` file:

```
{
  "preset": "laravel",
  "rules": {
    "simplified_null_return": true,
    "braces": false,
    "new_with_braces": {
      "anonymous_class": false,
      "named_class": false
    }
  }
}
```

Pint is built on top of [PHP-CS-Fixer](#). Therefore, you may use any of its rules to fix code style issues in your project: [PHP-CS-Fixer Configurator](#).

Excluding Files / Folders

By default, Pint will inspect all `.php` files in your project except those in the `vendor` directory. If you wish to exclude more folders, you may do so using the `exclude` configuration option:

```
{  
    "exclude": [  
        "my-specific/folder"  
    ]  
}
```

If you wish to exclude all files that contain a given name pattern, you may do so using the `notName` configuration option:

```
{  
    "notName": [  
        "*-my-file.php"  
    ]  
}
```

If you would like to exclude a file by providing an exact path to the file, you may do so using the `notPath` configuration option:

```
{  
    "notPath": [  
        "path/to/excluded-file.php"  
    ]  
}
```

Precognition

- [Introduction](#)
- [Live Validation](#)
 - [Using Vue](#)
 - [Using Vue and Inertia](#)
 - [Using React](#)
 - [Using React and Inertia](#)
 - [Using Alpine and Blade](#)
 - [Configuring Axios](#)
- [Customizing Validation Rules](#)
- [Handling File Uploads](#)
- [Managing Side-Effects](#)
- [Testing](#)

Introduction

Laravel Precognition allows you to anticipate the outcome of a future HTTP request. One of the primary use cases of Precognition is the ability to provide "live" validation for your frontend JavaScript application without having to duplicate your application's backend validation rules. Precognition pairs especially well with Laravel's Inertia-based [starter kits](#).

When Laravel receives a "precognitive request", it will execute all of the route's middleware and resolve the route's controller dependencies, including validating [form requests](#) – but it will not actually execute the route's controller method.

Live Validation

Using Vue

Using Laravel Precognition, you can offer live validation experiences to your users without having to duplicate your validation rules in your frontend Vue application. To illustrate how it works, let's build a form for creating new users within our application.

First, to enable Precognition for a route, the `HandlePrecognitiveRequests` middleware should be added to the route definition. You should also create a [form request](#) to house the route's validation rules:

```
use App\Http\Requests\StoreUserRequest;
use Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests;

Route::post('/users', function (StoreUserRequest $request) {
    // ...
})->middleware([HandlePrecognitiveRequests::class]);
```

Next, you should install the Laravel Precognition frontend helpers for Vue via NPM:

```
npm install laravel-precognition-vue
```

With the Laravel Precognition package installed, you can now create a form object using Precognition's `useForm` function, providing the HTTP method (`post`), the target URL (`/users`), and the initial form data.

Then, to enable live validation, invoke the form's `validate` method on each input's `change` event, providing the input's name:

```

<script setup>
import { useForm } from 'laravel-preognition-vue';

const form = useForm('post', '/users', {
    name: '',
    email: '',
});

const submit = () => form.submit();
</script>

<template>
<form @submit.prevent="submit">
    <label for="name">Name</label>
    <input
        id="name"
        v-model="form.name"
        @change="form.validate('name')"
    />
    <div v-if="form.invalid('name')">
        {{ form.errors.name }}
    </div>

    <label for="email">Email</label>
    <input
        id="email"
        type="email"
        v-model="form.email"
        @change="form.validate('email')"
    />
    <div v-if="form.invalid('email')">
        {{ form.errors.email }}
    </div>

    <button :disabled="form.processing">
        Create User
    </button>
</form>
</template>

```

Now, as the form is filled by the user, Precognition will provide live validation output powered by the validation rules in the route's form request. When the form's inputs are changed, a debounced "precognitive" validation request will be sent to your Laravel application. You may configure the debounce timeout by calling the form's `setValidationTimeout` function:

```
form.setValidationTimeout(3000);
```

When a validation request is in-flight, the form's `validating` property will be `true`:

```
<div v-if="form.validating">
    Validating...
</div>
```

Any validation errors returned during a validation request or a form submission will automatically populate the form's `errors` object:

```
<div v-if="form.invalid('email')">
  {{ form.errors.email }}
</div>
```

You can determine if the form has any errors using the form's `hasErrors` property:

```
<div v-if="form.hasErrors">
  <!-- ... -->
</div>
```

You may also determine if an input has passed or failed validation by passing the input's name to the form's `valid` and `invalid` functions, respectively:

```
<span v-if="form.valid('email')">
  ✓
</span>

<span v-else-if="form.invalid('email')">
  ✗
</span>
```



A form input will only appear as valid or invalid once it has changed and a validation response has been received.

If you are validating a subset of a form's inputs with Precognition, it can be useful to manually clear errors. You may use the form's `forgetError` function to achieve this:

```
<input
  id="avatar"
  type="file"
  @change="(e) => {
    form.avatar = e.target.files[0]

    form.forgetError('avatar')
  }"
>
```

Of course, you may also execute code in reaction to the response to the form submission. The form's `submit` function returns an Axios request promise. This provides a convenient way to access the response payload, reset the form inputs on successful submission, or handle a failed request:

```
const submit = () => form.submit()
  .then(response => {
    form.reset();

    alert('User created.');
  })
  .catch(error => {
    alert('An error occurred.');
  });
}
```

You may determine if a form submission request is in-flight by inspecting the form's `processing` property:

```
<button :disabled="form.processing">
  Submit
</button>
```

Using Vue and Inertia



If you would like a head start when developing your Laravel application with Vue and Inertia, consider using one of our [starter kits](#). Laravel's starter kits provide backend and frontend authentication scaffolding for your new Laravel application.

Before using Precognition with Vue and Inertia, be sure to review our general documentation on [using Precognition with Vue](#). When using Vue with Inertia, you will need to install the Inertia compatible Precognition library via NPM:

```
npm install laravel-precognition-vue-inertia
```

Once installed, Precognition's `useForm` function will return an Inertia [form helper](#) augmented with the validation features discussed above.

The form helper's `submit` method has been streamlined, removing the need to specify the HTTP method or URL. Instead, you may pass Inertia's [visit options](#) as the first and only argument. In addition, the `submit` method does not return a Promise as seen in the Vue example above. Instead, you may provide any of Inertia's supported [event callbacks](#) in the visit options given to the `submit` method:

```
<script setup>
import { useForm } from 'laravel-precognition-vue-inertia';

const form = useForm('post', '/users', {
  name: '',
  email: '',
});

const submit = () => form.submit({
  preserveScroll: true,
  onSuccess: () => form.reset(),
});
</script>
```

Using React

Using Laravel Precognition, you can offer live validation experiences to your users without having to duplicate your validation rules in your frontend React application. To illustrate how it works, let's build a form for creating new users within our application.

First, to enable Precognition for a route, the `HandlePrecognitiveRequests` middleware should be added to the route definition. You should also create a [form request](#) to house the route's validation rules:

```
use App\Http\Requests\StoreUserRequest;
use Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests;

Route::post('/users', function (StoreUserRequest $request) {
    // ...
})->middleware([HandlePrecognitiveRequests::class]);
```

Next, you should install the Laravel Precognition frontend helpers for React via NPM:

```
npm install laravel-precognition-react
```

With the Laravel Precognition package installed, you can now create a form object using Precognition's `useForm` function, providing the HTTP method (`post`), the target URL (`/users`), and the initial form data.

To enable live validation, you should listen to each input's `change` and `blur` event. In the `change` event handler, you should set the form's data with the `setData` function, passing the input's name and new value. Then, in the `blur` event handler invoke the form's `validate` method, providing the input's name:

```

import { useForm } from 'laravel-preognition-react';

export default function Form() {
    const form = useForm('post', '/users', {
        name: '',
        email: '',
    });

    const submit = (e) => {
        e.preventDefault();

        form.submit();
    };

    return (
        <form onSubmit={submit}>
            <label for="name">Name</label>
            <input
                id="name"
                value={form.data.name}
                onChange={(e) => form.setData('name', e.target.value)}
                onBlur={() => form.validate('name')}
            />
            {form.invalid('name') && <div>{form.errors.name}</div>

            <label for="email">Email</label>
            <input
                id="email"
                value={form.data.email}
                onChange={(e) => form.setData('email', e.target.value)}
                onBlur={() => form.validate('email')}
            />
            {form.invalid('email') && <div>{form.errors.email}</div>

            <button disabled={form.processing}>
                Create User
            </button>
        </form>
    );
}

```

Now, as the form is filled by the user, Precognition will provide live validation output powered by the validation rules in the route's form request. When the form's inputs are changed, a debounced "precognitive" validation request will be sent to your Laravel application. You may configure the debounce timeout by calling the form's `setValidationTimeout` function:

```
form.setValidationTimeout(3000);
```

When a validation request is in-flight, the form's `validating` property will be `true`:

```
{form.validating && <div>Validating...</div>}
```

Any validation errors returned during a validation request or a form submission will automatically populate the form's `errors` object:

```
{form.invalid('email') && <div>{form.errors.email}</div>}
```

You can determine if the form has any errors using the form's `hasErrors` property:

```
{form.hasErrors && <div><!-- ... --></div>}
```

You may also determine if an input has passed or failed validation by passing the input's name to the form's `valid` and `invalid` functions, respectively:

```
{form.valid('email') && <span>✓</span>}
```

```
{form.invalid('email') && <span>✗</span>}
```



A form input will only appear as valid or invalid once it has changed and a validation response has been received.

If you are validating a subset of a form's inputs with Precognition, it can be useful to manually clear errors. You may use the form's `forgetError` function to achieve this:

```
<input
    id="avatar"
    type="file"
    onChange={(e) =>
        form.setData('avatar', e.target.value);

        form.forgetError('avatar');
    }
>
```

Of course, you may also execute code in reaction to the response to the form submission. The form's `submit` function returns an Axios request promise. This provides a convenient way to access the response payload, reset the form's inputs on a successful form submission, or handle a failed request:

```
const submit = (e) => {
    e.preventDefault();

    form.submit()
        .then(response => {
            form.reset();

            alert('User created.');
        })
        .catch(error => {
            alert('An error occurred.');
        });
};
```

You may determine if a form submission request is in-flight by inspecting the form's `processing` property:

```
<button disabled={form.processing}>
    Submit
</button>
```

Using React and Inertia



If you would like a head start when developing your Laravel application with React and Inertia, consider using one of our [starter kits](#). Laravel's starter kits provide backend and frontend authentication scaffolding for your new Laravel application.

Before using Precognition with React and Inertia, be sure to review our general documentation on [using Precognition with React](#). When using React with Inertia, you will need to install the Inertia compatible Precognition library via NPM:

```
npm install laravel-precognition-react-inertia
```

Once installed, Precognition's `useForm` function will return an Inertia [form helper](#) augmented with the validation features discussed above.

The form helper's `submit` method has been streamlined, removing the need to specify the HTTP method or URL. Instead, you may pass Inertia's [visit options](#) as the first and only argument. In addition, the `submit` method does not return a Promise as seen in the React example above. Instead, you may provide any of Inertia's supported [event callbacks](#) in the visit options given to the `submit` method:

```
import { useForm } from 'laravel-precognition-react-inertia';

const form = useForm('post', '/users', {
    name: '',
    email: '',
});

const submit = (e) => {
    e.preventDefault();

    form.submit({
        preserveScroll: true,
        onSuccess: () => form.reset(),
    });
};
```

Using Alpine and Blade

Using Laravel Precognition, you can offer live validation experiences to your users without having to duplicate your validation rules in your frontend Alpine application. To illustrate how it works, let's build a form for creating new users within our application.

First, to enable Precognition for a route, the `HandlePrecognitiveRequests` middleware should be added to the route definition. You should also create a [form request](#) to house the route's validation rules:

```
use App\Http\Requests\CreateUserRequest;
use Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests;

Route::post('/users', function (CreateUserRequest $request) {
    // ...
})->middleware([HandlePrecognitiveRequests::class]);
```

Next, you should install the Laravel Precognition frontend helpers for Alpine via NPM:

```
npm install laravel-precognition-alpine
```

Then, register the Precognition plugin with Alpine in your `resources/js/app.js` file:

```
import Alpine from 'alpinejs';
import Precognition from 'laravel-precognition-alpine';

window.Alpine = Alpine;

Alpine.plugin(Precognition);
Alpine.start();
```

With the Laravel Precognition package installed and registered, you can now create a form object using Precognition's `$form` "magic", providing the HTTP method (`post`), the target URL (`/users`), and the initial form data.

To enable live validation, you should bind the form's data to its relevant input and then listen to each input's `change` event. In the `change` event handler, you should invoke the form's `validate` method, providing the input's name:

```

<form x-data="{
    form: $form('post', '/register', {
        name: '',
        email: '',
    }),
}>
    @csrf
    <label for="name">Name</label>
    <input
        id="name"
        name="name"
        x-model="form.name"
        @change="form.validate('name')"
    />
    <template x-if="form.invalid('name')">
        <div x-text="form.errors.name"></div>
    </template>

    <label for="email">Email</label>
    <input
        id="email"
        name="email"
        x-model="form.email"
        @change="form.validate('email')"
    />
    <template x-if="form.invalid('email')">
        <div x-text="form.errors.email"></div>
    </template>

    <button :disabled="form.processing">
        Create User
    </button>
</form>

```

Now, as the form is filled by the user, Precognition will provide live validation output powered by the validation rules in the route's form request. When the form's inputs are changed, a debounced "precognitive" validation request will be sent to your Laravel application. You may configure the debounce timeout by calling the form's `setValidationTimeout` function:

```
form.setValidationTimeout(3000);
```

When a validation request is in-flight, the form's `validating` property will be `true`:

```
<template x-if="form.validating">
    <div>Validating...</div>
</template>
```

Any validation errors returned during a validation request or a form submission will automatically populate the form's `errors` object:

```
<template x-if="form.invalid('email')">
    <div x-text="form.errors.email"></div>
</template>
```

You can determine if the form has any errors using the form's `hasErrors` property:

```
<template x-if="form.hasErrors">
  <div><!-- ... --></div>
</template>
```

You may also determine if an input has passed or failed validation by passing the input's name to the form's `valid` and `invalid` functions, respectively:

```
<template x-if="form.valid('email')">
  <span>✓ </span>
</template>

<template x-if="form.invalid('email')">
  <span>✗ </span>
</template>
```



A form input will only appear as valid or invalid once it has changed and a validation response has been received.

You may determine if a form submission request is in-flight by inspecting the form's `processing` property:

```
<button :disabled="form.processing">
  Submit
</button>
```

Repopulating Old Form Data

In the user creation example discussed above, we are using Precognition to perform live validation; however, we are performing a traditional server-side form submission to submit the form. So, the form should be populated with any "old" input and validation errors returned from the server-side form submission:

```
<form x-data="{
  form: $form('post', '/register', {
    name: '{{ old('name') }}',
    email: '{{ old('email') }}',
  }).setErrors('{{ Js::from($errors->messages()) }}),
}">
```

Alternatively, if you would like to submit the form via XHR you may use the form's `submit` function, which returns an Axios request promise:

```
<form
  x-data="{
    form: $form('post', '/register', {
      name: '',
      email: '',
    }),
    submit() {
      this.form.submit()
        .then(response => {
          form.reset();

          alert('User created.')
        })
        .catch(error => {
          alert('An error occurred.');
        });
    },
  }"
  @submit.prevent="submit"
>
```

Configuring Axios

The Precognition validation libraries use the [Axios](#) HTTP client to send requests to your application's backend. For convenience, the Axios instance may be customized if required by your application. For example, when using the `laravel-precognition-vue` library, you may add additional request headers to each outgoing request in your application's `resources/js/app.js` file:

```
import { client } from 'laravel-precognition-vue';

client.axios().defaults.headers.common['Authorization'] = authToken;
```

Or, if you already have a configured Axios instance for your application, you may tell Precognition to use that instance instead:

```
import Axios from 'axios';
import { client } from 'laravel-precognition-vue';

window.axios = Axios.create()
window.axios.defaults.headers.common['Authorization'] = authToken;

client.use(window.axios)
```



The Inertia flavored Precognition libraries will only use the configured Axios instance for validation requests. Form submissions will always be sent by Inertia.

Customizing Validation Rules

It is possible to customize the validation rules executed during a precognitive request by using the request's `isPrecognitive` method.

For example, on a user creation form, we may want to validate that a password is "uncompromised" only on the final form submission. For precognitive validation requests, we will simply validate that the password is required and has a minimum of 8 characters. Using the `isPrecognitive` method, we can customize the rules defined by our form request:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rules\Password;

class StoreUserRequest extends FormRequest
{
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    protected function rules()
    {
        return [
            'password' => [
                'required',
                $this->isPrecognitive()
                    ? Password::min(8)
                    : Password::min(8)->uncompromised(),
            ],
            // ...
        ];
    }
}
```

Handling File Uploads

By default, Laravel Precognition does not upload or validate files during a precognitive validation request. This ensure that large files are not unnecessarily uploaded multiple times.

Because of this behavior, you should ensure that your application [customizes the corresponding form request's validation rules](#) to specify the field is only required for full form submissions:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
protected function rules()
{
    return [
        'avatar' => [
            ...$this->isPrecognitive() ? [] : ['required'],
            'image',
            'mimes:jpg,png',
            'dimensions:ratio=3/2',
        ],
        // ...
    ];
}
```

If you would like to include files in every validation request, you may invoke the `validateFiles` function on your client-side form instance:

```
form.validateFiles();
```

Managing Side-Effects

When adding the `HandlePrecognitiveRequests` middleware to a route, you should consider if there are any side-effects in *other* middleware that should be skipped during a precognitive request.

For example, you may have a middleware that increments the total number of "interactions" each user has with your application, but you may not want precognitive requests to be counted as an interaction. To accomplish this, we may check the request's `isPrecognitive` method before incrementing the interaction count:

```
<?php

namespace App\Http\Middleware;

use App\Facades\Interaction;
use Closure;
use Illuminate\Http\Request;

class InteractionMiddleware
{
    /**
     * Handle an incoming request.
     */
    public function handle(Request $request, Closure $next): mixed
    {
        if (! $request->isPrecognitive()) {
            Interaction::incrementFor($request->user());
        }

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

Testing

If you would like to make precognitive requests in your tests, Laravel's `TestCase` includes a `withPrecognition` helper which will add the `Precognition` request header.

Additionally, if you would like to assert that a precognitive request was successful, e.g., did not return any validation errors, you may use the `assertSuccessfulPrecognition` method on the response:

```
public function test_it_validates_registration_form_with_precognition()
{
    $response = $this->withPrecognition()
        ->post('/register', [
            'name' => 'Taylor Otwell',
        ]);

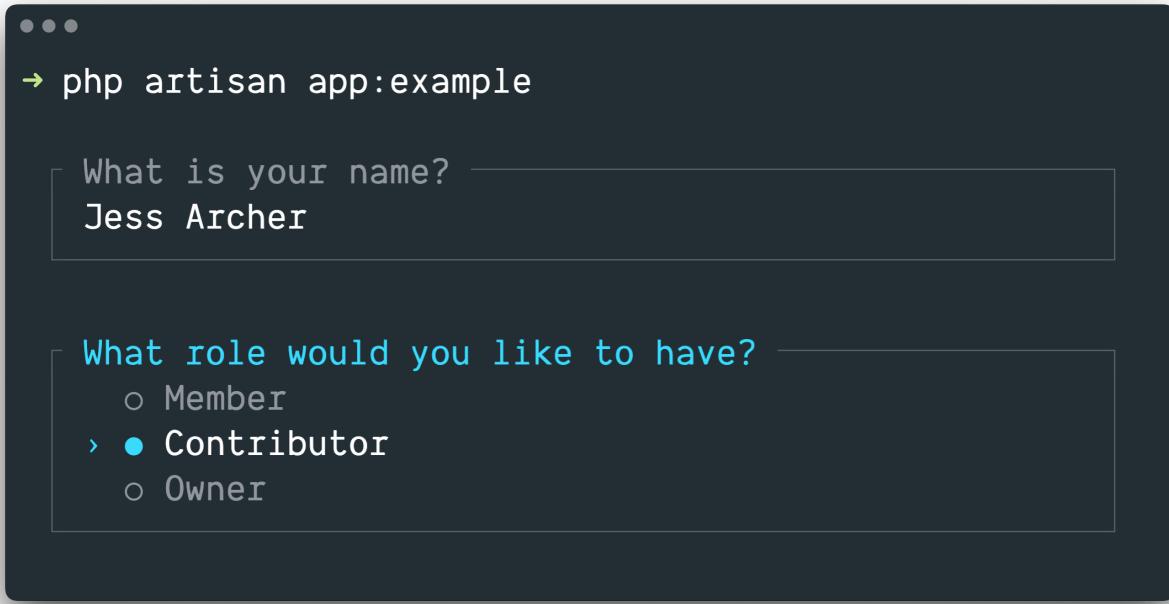
    $response->assertSuccessfulPrecognition();
    $this->assertSame(0, User::count());
}
```

Prompts

- [Introduction](#)
- [Installation](#)
- [Available Prompts](#)
 - [Text](#)
 - [Password](#)
 - [Confirm](#)
 - [Select](#)
 - [Multi-select](#)
 - [Suggest](#)
 - [Search](#)
 - [Multi-search](#)
 - [Pause](#)
- [Informational Messages](#)
- [Tables](#)
- [Spin](#)
- [Progress Bar](#)
- [Terminal Considerations](#)
- [Unsupported Environments and Fallbacks](#)

Introduction

[Laravel Prompts](#) is a PHP package for adding beautiful and user-friendly forms to your command-line applications, with browser-like features including placeholder text and validation.



Laravel Prompts is perfect for accepting user input in your [Artisan console commands](#), but it may also be used in any command-line PHP project.



Laravel Prompts supports macOS, Linux, and Windows with WSL. For more information, please see our documentation on [unsupported environments & fallbacks](#).

Installation

Laravel Prompts is already included with the latest release of Laravel.

Laravel Prompts may also be installed in your other PHP projects by using the Composer package manager:

```
composer require laravel/prompts
```

Available Prompts

Text

The `text` function will prompt the user with the given question, accept their input, and then return it:

```
use function Laravel\Prompts\text;

$name = text('What is your name?');
```

You may also include placeholder text, a default value, and an informational hint:

```
$name = text(
    label: 'What is your name?',
    placeholder: 'E.g. Taylor Otwell',
    default: $user?->name,
    hint: 'This will be displayed on your profile.'
);
```

Required Values

If you require a value to be entered, you may pass the `required` argument:

```
$name = text(
    label: 'What is your name?',
    required: true
);
```

If you would like to customize the validation message, you may also pass a string:

```
$name = text(
    label: 'What is your name?',
    required: 'Your name is required.'
);
```

Additional Validation

Finally, if you would like to perform additional validation logic, you may pass a closure to the `validate` argument:

```
$name = text(
    label: 'What is your name?',
    validate: fn (string $value) => match ($true) {
        strlen($value) < 3 => 'The name must be at least 3 characters.',
        strlen($value) > 255 => 'The name must not exceed 255 characters.',
        default => null
    }
);
```

The closure will receive the value that has been entered and may return an error message, or `null` if the validation passes.

Password

The `password` function is similar to the `text` function, but the user's input will be masked as they type in the console. This is useful when asking for sensitive information such as passwords:

```
use function Laravel\Prompts\password;

$password = password('What is your password?');
```

You may also include placeholder text and an informational hint:

```
$password = password(
    label: 'What is your password?',
    placeholder: 'password',
    hint: 'Minimum 8 characters.'
);
```

Required Values

If you require a value to be entered, you may pass the `required` argument:

```
$password = password(
    label: 'What is your password?',
    required: true
);
```

If you would like to customize the validation message, you may also pass a string:

```
$password = password(
    label: 'What is your password?',
    required: 'The password is required.'
);
```

Additional Validation

Finally, if you would like to perform additional validation logic, you may pass a closure to the `validate` argument:

```
$password = password(
    label: 'What is your password?',
    validate: fn (string $value) => match ($value) {
        strlen($value) < 8 => 'The password must be at least 8 characters.',
        default => null
    }
);
```

The closure will receive the value that has been entered and may return an error message, or `null` if the validation passes.

Confirm

If you need to ask the user for a "yes or no" confirmation, you may use the `confirm` function. Users may use the arrow keys or press `y` or `n` to select their response. This function will return either `true` or `false`.

```
use function Laravel\Prompts\confirm;

$confirmed = confirm('Do you accept the terms?');
```

You may also include a default value, customized wording for the "Yes" and "No" labels, and an informational hint:

```
$confirmed = confirm(
    label: 'Do you accept the terms?',
    default: false,
    yes: 'I accept',
    no: 'I decline',
    hint: 'The terms must be accepted to continue.'
);
```

Requiring "Yes"

If necessary, you may require your users to select "Yes" by passing the `required` argument:

```
$confirmed = confirm(
    label: 'Do you accept the terms?',
    required: true
);
```

If you would like to customize the validation message, you may also pass a string:

```
$confirmed = confirm(
    label: 'Do you accept the terms?',
    required: 'You must accept the terms to continue.'
);
```

Select

If you need the user to select from a predefined set of choices, you may use the `select` function:

```
use function Laravel\Prompts\select;

$role = select(
    'What role should the user have?',
    ['Member', 'Contributor', 'Owner'],
);
```

You may also specify the default choice and an informational hint:

```
$role = select(
    label: 'What role should the user have?',
    options: ['Member', 'Contributor', 'Owner'],
    default: 'Owner',
    hint: 'The role may be changed at any time.'
);
```

You may also pass an associative array to the `options` argument to have the selected key returned instead of its value:

```
$role = select(
    label: 'What role should the user have?',
    options: [
        'member' => 'Member',
        'contributor' => 'Contributor',
        'owner' => 'Owner'
    ],
    default: 'owner'
);
```

Up to five options will be displayed before the list begins to scroll. You may customize this by passing the `scroll` argument:

```
$role = select(
    label: 'Which category would you like to assign?',
    options: Category::pluck('name', 'id'),
    scroll: 10
);
```

Validation

Unlike other prompt functions, the `select` function doesn't accept the `required` argument because it is not possible to select nothing. However, you may pass a closure to the `validate` argument if you need to present an option but prevent it from being selected:

```
$role = select(
    label: 'What role should the user have?',
    options: [
        'member' => 'Member',
        'contributor' => 'Contributor',
        'owner' => 'Owner'
    ],
    validate: fn (string $value) =>
        $value === 'owner' && User::where('role', 'owner')->exists()
            ? 'An owner already exists.'
            : null
);
```

If the `options` argument is an associative array, then the closure will receive the selected key, otherwise it will receive the selected value. The closure may return an error message, or `null` if the validation passes.

Multi-select

If you need to the user to be able to select multiple options, you may use the `multiselect` function:

```
use function Laravel\Prompts\multiselect;

$permissions = multiselect(
    'What permissions should be assigned?',
    ['Read', 'Create', 'Update', 'Delete']
);
```

You may also specify default choices and an informational hint:

```
use function Laravel\Prompts\multiselect;

$permissions = multiselect(
    label: 'What permissions should be assigned?',
    options: ['Read', 'Create', 'Update', 'Delete'],
    default: ['Read', 'Create'],
    hint: 'Permissions may be updated at any time.'
);
```

You may also pass an associative array to the `options` argument to return the selected options' keys instead of their values:

```
$permissions = multiselect(
    label: 'What permissions should be assigned?',
    options: [
        'read' => 'Read',
        'create' => 'Create',
        'update' => 'Update',
        'delete' => 'Delete'
    ],
    default: ['read', 'create']
);
```

Up to five options will be displayed before the list begins to scroll. You may customize this by passing the `scroll` argument:

```
$categories = multiselect(
    label: 'What categories should be assigned?',
    options: Category::pluck('name', 'id'),
    scroll: 10
);
```

Requiring a Value

By default, the user may select zero or more options. You may pass the `required` argument to enforce one or more options instead:

```
$categories = multiselect(
    label: 'What categories should be assigned?',
    options: Category::pluck('name', 'id'),
    required: true,
);
```

If you would like to customize the validation message, you may provide a string to the `required` argument:

```
$categories = multiselect(
    label: 'What categories should be assigned?',
    options: Category::pluck('name', 'id'),
    required: 'You must select at least one category',
);
```

Validation

You may pass a closure to the `validate` argument if you need to present an option but prevent it from being selected:

```
$permissions = multiselect(
    label: 'What permissions should the user have?',
    options: [
        'read' => 'Read',
        'create' => 'Create',
        'update' => 'Update',
        'delete' => 'Delete'
    ],
    validate: fn (array $values) => ! in_array('read', $values)
        ? 'All users require the read permission.'
        : null
);
```

If the `options` argument is an associative array then the closure will receive the selected keys, otherwise it will receive the selected values. The closure may return an error message, or `null` if the validation passes.

Suggest

The `suggest` function can be used to provide auto-completion for possible choices. The user can still provide any answer, regardless of the auto-completion hints:

```
use function Laravel\Prompts\suggest;

$name = suggest('What is your name?', ['Taylor', 'Dayle']);
```

Alternatively, you may pass a closure as the second argument to the `suggest` function. The closure will be called each time the user types an input character. The closure should accept a string parameter containing the user's input so far and return an array of options for auto-completion:

```
$name = suggest(
    'What is your name?',
    fn ($value) => collect(['Taylor', 'Dayle'])
        ->filter(fn ($name) => Str::contains($name, $value, ignoreCase: true))
)
```

You may also include placeholder text, a default value, and an informational hint:

```
$name = suggest(
    label: 'What is your name?',
    options: ['Taylor', 'Dayle'],
    placeholder: 'E.g. Taylor',
    default: $user?->name,
    hint: 'This will be displayed on your profile.'
);
```

Required Values

If you require a value to be entered, you may pass the `required` argument:

```
$name = suggest(
    label: 'What is your name?',
    options: ['Taylor', 'Dayle'],
    required: true
);
```

If you would like to customize the validation message, you may also pass a string:

```
$name = suggest(
    label: 'What is your name?',
    options: ['Taylor', 'Dayle'],
    required: 'Your name is required.'
);
```

Additional Validation

Finally, if you would like to perform additional validation logic, you may pass a closure to the `validate` argument:

```
$name = suggest(
    label: 'What is your name?',
    options: ['Taylor', 'Dayle'],
    validate: fn (string $value) => match ($value) {
        strlen($value) < 3 => 'The name must be at least 3 characters.',
        strlen($value) > 255 => 'The name must not exceed 255 characters.',
        default => null
    }
);
```

The closure will receive the value that has been entered and may return an error message, or `null` if the validation passes.

Search

If you have a lot of options for the user to select from, the `search` function allows the user to type a search query to filter the results before using the arrow keys to select an option:

```
use function Laravel\Prompts\search;

$id = search(
    'Search for the user that should receive the mail',
    fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : []
);
```

The closure will receive the text that has been typed by the user so far and must return an array of options. If you return an associative array then the selected option's key will be returned, otherwise its value will be returned instead.

You may also include placeholder text and an informational hint:

```
$id = search(
    label: 'Search for the user that should receive the mail',
    placeholder: 'E.g. Taylor Otwell',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    hint: 'The user will receive an email immediately.'
);
```

Up to five options will be displayed before the list begins to scroll. You may customize this by passing the `scroll` argument:

```
$id = search(
    label: 'Search for the user that should receive the mail',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    scroll: 10
);
```

Validation

If you would like to perform additional validation logic, you may pass a closure to the `validate` argument:

```
$id = search(
    label: 'Search for the user that should receive the mail',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    validate: function (int|string $value) {
        $user = User::findOrFail($value);

        if ($user->opted_out) {
            return 'This user has opted-out of receiving mail.';
        }
    }
);
```

If the `options` closure returns an associative array, then the closure will receive the selected key, otherwise, it will receive the selected value. The closure may return an error message, or `null` if the validation passes.

Multi-search

If you have a lot of searchable options and need the user to be able to select multiple items, the `multisearch` function allows the user to type a search query to filter the results before using the arrow keys and space-bar to select options:

```
use function Laravel\Prompts\multisearch;

$ids = multisearch(
    'Search for the users that should receive the mail',
    fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : []
);
```

The closure will receive the text that has been typed by the user so far and must return an array of options. If you return an associative array then the selected options' keys will be returned; otherwise, their values will be returned instead.

You may also include placeholder text and an informational hint:

```
$ids = multisearch(
    label: 'Search for the users that should receive the mail',
    placeholder: 'E.g. Taylor Otwell',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    hint: 'The user will receive an email immediately.'
);
```

Up to five options will be displayed before the list begins to scroll. You may customize this by providing the `scroll` argument:

```
$ids = multiselect(
    label: 'Search for the users that should receive the mail',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    scroll: 10
);
```

Requiring a Value

By default, the user may select zero or more options. You may pass the `required` argument to enforce one or more options instead:

```
$ids = multiselect(
    'Search for the users that should receive the mail',
    fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    required: true,
);
```

If you would like to customize the validation message, you may also provide a string to the `required` argument:

```
$ids = multiselect(
    'Search for the users that should receive the mail',
    fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    required: 'You must select at least one user.'
);
```

Validation

If you would like to perform additional validation logic, you may pass a closure to the `validate` argument:

```
$ids = multiselect(
    label: 'Search for the users that should receive the mail',
    options: fn (string $value) => strlen($value) > 0
        ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
        : [],
    validate: function (array $values) {
        $optedOut = User::where('name', 'like', '%a%')->findMany($values);

        if ($optedOut->isNotEmpty()) {
            return $optedOut->pluck('name')->join(', ', ', and ') . ' have opted out.';
        }
    }
);
```

If the `options` closure returns an associative array, then the closure will receive the selected keys; otherwise, it will receive the selected values. The closure may return an error message, or `null` if the validation passes.

Pause

The `pause` function may be used to display informational text to the user and wait for them to confirm their desire to proceed by pressing the Enter / Return key:

```
use function Laravel\Prompts\pause;

pause('Press ENTER to continue.');
```

Informational Messages

The `note`, `info`, `warning`, `error`, and `alert` functions may be used to display informational messages:

```
use function Laravel\Prompts\info;

info('Package installed successfully.');
```

Tables

The `table` function makes it easy to display multiple rows and columns of data. All you need to do is provide the column names and the data for the table:

```
use function Laravel\Prompts\table;

table(
    ['Name', 'Email'],
    User::all(['name', 'email'])
);
```

Spin

The `spin` function displays a spinner along with an optional message while executing a specified callback. It serves to indicate ongoing processes and returns the callback's results upon completion:

```
use function Laravel\Prompts\spin;

$response = spin(
    fn () => Http::get('http://example.com'),
    'Fetching response...'
);
```



The `spin` function requires the `pcntl` PHP extension to animate the spinner. When this extension is not available, a static version of the spinner will appear instead.

Progress Bars

For long running tasks, it can be helpful to show a progress bar that informs users how complete the task is. Using the `progress` function, Laravel will display a progress bar and advance its progress for each iteration over a given iterable value:

```
use function Laravel\Prompts\progress;

$users = progress(
    label: 'Updating users',
    steps: User::all(),
    callback: fn ($user) => $this->performTask($user),
);
```

The `progress` function acts like a map function and will return an array containing the return value of each iteration of your callback.

The callback may also accept the `\Laravel\Prompts\Progress` instance, allowing you to modify the label and hint on each iteration:

```
$users = progress(
    label: 'Updating users',
    steps: User::all(),
    callback: function ($user, $progress) {
        $progress
            ->label("Updating {$user->name}")
            ->hint("Created on {$user->created_at}");

        return $this->performTask($user);
    },
    hint: 'This may take some time.',
);
```

Sometimes, you may need more manual control over how a progress bar is advanced. First, define the total number of steps the process will iterate through. Then, advance the progress bar via the `advance` method after processing each item:

```
$progress = progress(label: 'Updating users', steps: 10);

$users = User::all();

$progress->start();

foreach ($users as $user) {
    $this->performTask($user);

    $progress->advance();
}

$progress->finish();
```

Terminal Considerations

Terminal Width

If the length of any label, option, or validation message exceeds the number of "columns" in the user's terminal, it will be automatically truncated to fit. Consider minimizing the length of these strings if your users may be using narrower terminals. A typically safe maximum length is 74 characters to support an 80-character terminal.

Terminal Height

For any prompts that accept the `scroll` argument, the configured value will automatically be reduced to fit the height of the user's terminal, including space for a validation message.

Unsupported Environments and Fallbacks

Laravel Prompts supports macOS, Linux, and Windows with WSL. Due to limitations in the Windows version of PHP, it is not currently possible to use Laravel Prompts on Windows outside of WSL.

For this reason, Laravel Prompts supports falling back to an alternative implementation such as the [Symfony Console Question Helper](#).



When using Laravel Prompts with the Laravel framework, fallbacks for each prompt have been configured for you and will be automatically enabled in unsupported environments.

Fallback Conditions

If you are not using Laravel or need to customize when the fallback behavior is used, you may pass a boolean to the `fallbackWhen` static method on the `Prompt` class:

```
use Laravel\Prompts\Prompt;

Prompt::fallbackWhen(
    ! $input->isInteractive() || windows_os() || app()->runningUnitTests()
);
```

Fallback Behavior

If you are not using Laravel or need to customize the fallback behavior, you may pass a closure to the `fallbackUsing` static method on each prompt class:

```
use Laravel\Prompts\TextPrompt;
use Symfony\Component\Console\Question\Question;
use Symfony\Component\Console\Style\SymfonyStyle;

TextPrompt::fallbackUsing(function (TextPrompt $prompt) use ($input, $output) {
    $question = (new Question($prompt->label, $prompt->default ?: null))
        ->setValidator(function ($answer) use ($prompt) {
            if ($prompt->required && $answer === null) {
                throw new \RuntimeException(is_string($prompt->required) ? $prompt->required :
'Required.');
            }

            if ($prompt->validate) {
                $error = ($prompt->validate)($answer ?? '');
            }

            if ($error) {
                throw new \RuntimeException($error);
            }
        });

        return $answer;
});

return (new SymfonyStyle($input, $output))
    ->askQuestion($question);
});
```

Fallbacks must be configured individually for each prompt class. The closure will receive an instance of the prompt class and must return an appropriate type for the prompt.

Laravel Pulse

- [Introduction](#)
- [Installation](#)
 - [Configuration](#)
- [Dashboard](#)
 - [Authorization](#)
 - [Customization](#)
 - [Resolving Users](#)
 - [Cards](#)
- [Capturing Entries](#)
 - [Recorders](#)
 - [Filtering](#)
- [Performance](#)
 - [Using a Different Database](#)
 - [Redis Ingest](#)
 - [Sampling](#)
 - [Trimming](#)
 - [Handling Pulse Exceptions](#)
- [Custom Cards](#)
 - [Card Components](#)
 - [Styling](#)
 - [Data Capture and Aggregation](#)

Introduction

[Laravel Pulse](#) delivers at-a-glance insights into your application's performance and usage. With Pulse, you can track down bottlenecks like slow jobs and endpoints, find your most active users, and more.

For in-depth debugging of individual events, check out [Laravel Telescope](#).

Installation



Pulse's first-party storage implementation currently requires a MySQL or PostgreSQL database. If you are using a different database engine, you will need a separate MySQL or PostgreSQL database for your Pulse data.

Since Pulse is currently in beta, you may need to adjust your application's `composer.json` file to allow beta package releases to be installed:

```
"minimum-stability": "beta",
"prefer-stable": true
```

Then, you may use the Composer package manager to install Pulse into your Laravel project:

```
composer require laravel/pulse
```

Next, you should publish the Pulse configuration and migration files using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --provider="Laravel\Pulse\PulseServiceProvider"
```

Finally, you should run the `migrate` command in order to create the tables needed to store Pulse's data:

```
php artisan migrate
```

Once Pulse's database migrations have been run, you may access the Pulse dashboard via the `/pulse` route.



If you do not want to store Pulse data in your application's primary database, you may [specify a dedicated database connection](#).

Configuration

Many of Pulse's configuration options can be controlled using environment variables. To see the available options, register new recorders, or configure advanced options, you may publish the `config/pulse.php` configuration file:

```
php artisan vendor:publish --tag=pulse-config
```

Dashboard

Authorization

The Pulse dashboard may be accessed via the `/pulse` route. By default, you will only be able to access this dashboard in the `local` environment, so you will need to configure authorization for your production environments by customizing the `'viewPulse'` authorization gate. You can accomplish this within your application's `app/Providers/AuthServiceProvider.php` file:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Gate::define('viewPulse', function (User $user) {
        return $user->isAdmin();
    });

    // ...
}
```

Customization

The Pulse dashboard cards and layout may be configured by publishing the dashboard view. The dashboard view will be published to `resources/views/vendor/pulse/dashboard.blade.php`:

```
php artisan vendor:publish --tag=pulse-dashboard
```

The dashboard is powered by [Livewire](#), and allows you to customize the cards and layout without needing to rebuild any JavaScript assets.

Within this file, the `<x-pulse>` component is responsible for rendering the dashboard and provides a grid layout for the cards. If you would like the dashboard to span the full width of the screen, you may provide the `full-width` prop to the component:

```
<x-pulse full-width>
  ...
</x-pulse>
```

By default, the `<x-pulse>` component will create a 12 column grid, but you may customize this using the `cols` prop:

```
<x-pulse cols="16">
  ...
</x-pulse>
```

Each card accepts a `cols` and `rows` prop to control the space and positioning:

```
<livewire:pulse.usage cols="4" rows="2" />
```

Most cards also accept an `expand` prop to show the full card instead of scrolling:

```
<livewire:pulse.slow-queries expand />
```

Resolving Users

For cards that display information about your users, such as the Application Usage card, Pulse will only record the user's ID. When rendering the dashboard, Pulse will resolve the `name` and `email` fields from your default `Authenticatable` model and display avatars using the Gravatar web service.

You may customize the fields and avatar by invoking the `Pulse::user` method within your application's `App\Providers\AppServiceProvider` class.

The `user` method accepts a closure which will receive the `Authenticatable` model to be displayed and should return an array containing `name`, `extra`, and `avatar` information for the user:

```
use Laravel\Pulse\Facades\Pulse;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Pulse::user(fn ($user) => [
        'name' => $user->name,
        'extra' => $user->email,
        'avatar' => $user->avatar_url,
    ]);

    // ...
}
```



You may completely customize how the authenticated user is captured and retrieved by implementing the `Laravel\Pulse\Contracts\ResolvesUsers` contract and binding it in Laravel's [service container](#).

Cards

Servers

The `<livewire:pulse.servers />` card displays system resource usage for all servers running the `pulse:check` command. Please refer to the documentation regarding the [servers recorder](#) for more information on system resource reporting.

Application Usage

The `<livewire:pulse.usage />` card displays the top 10 users making requests to your application, dispatching jobs, and experiencing slow requests.

If you wish to view all usage metrics on screen at the same time, you may include the card multiple times and specify the `type` attribute:

```
<livewire:pulse.usage type="requests" />
<livewire:pulse.usage type="slow_requests" />
<livewire:pulse.usage type="jobs" />
```

To learn how to customize how Pulse retrieves and displays user information, consult our documentation on [resolving users](#).



If your application receives a lot of requests or dispatches a lot of jobs, you may wish to enable [sampling](#). See the [user requests recorder](#), [user jobs recorder](#), and [slow jobs recorder](#) documentation for more information.

Exceptions

The `<livewire:pulse.exceptions />` card shows the frequency and recency of exceptions occurring in your application. By default, exceptions are grouped based on the exception class and location where it occurred. See the [exceptions recorder](#) documentation for more information.

Queues

The `<livewire:pulse.queues />` card shows the throughput of the queues in your application, including the number of jobs queued, processing, processed, released, and failed. See the [queues recorder](#) documentation for more information.

Slow Requests

The `<livewire:pulse.slow-requests />` card shows incoming requests to your application that exceed the configured threshold, which is 1,000ms by default. See the [slow requests recorder](#) documentation for more information.

Slow Jobs

The `<livewire:pulse.slow-jobs />` card shows the queued jobs in your application that exceed the configured threshold, which is 1,000ms by default. See the [slow jobs recorder](#) documentation for more information.

Slow Queries

The `<livewire:pulse.slow-queries />` card shows the database queries in your application that exceed the configured threshold, which is 1,000ms by default.

By default, slow queries are grouped based on the SQL query (without bindings) and the location where it occurred, but you may choose to not capture the location if you wish to group solely on the SQL query.

See the [slow queries recorder](#) documentation for more information.

Slow Outgoing Requests

The `<livewire:pulse.slow-outgoing-requests />` card shows outgoing requests made using Laravel's [HTTP client](#) that exceed the configured threshold, which is 1,000ms by default.

By default, entries will be grouped by the full URL. However, you may wish to normalize or group similar outgoing requests using regular expressions. See the [slow outgoing requests recorder](#) documentation for more information.

Cache

The `<livewire:pulse.cache />` card shows the cache hit and miss statistics for your application, both globally and for individual keys.

By default, entries will be grouped by key. However, you may wish to normalize or group similar keys using regular expressions. See the [cache interactions recorder](#) documentation for more information.

Capturing Entries

Most Pulse recorders will automatically capture entries based on framework events dispatched by Laravel. However, the [servers recorder](#) and some third-party cards must poll for information regularly. To use these cards, you must run the `pulse:check` daemon on all of your individual application servers:

```
php artisan pulse:check
```



To keep the `pulse:check` process running permanently in the background, you should use a process monitor such as Supervisor to ensure that the command does not stop running.

As the `pulse:check` command is a long-lived process, it will not see changes to your codebase without being restarted. You should gracefully restart the command by calling the `pulse:restart` command during your application's deployment process:

```
php artisan pulse:restart
```



Pulse uses the `cache` to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Recorders

Recorders are responsible for capturing entries from your application to be recorded in the Pulse database. Recorders are registered and configured in the `recorders` section of the [Pulse configuration file](#).

Cache Interactions

The `CacheInteractions` recorder captures information about the `cache` hits and misses occurring in your application for display on the `Cache` card.

You may optionally adjust the `sample rate` and ignored key patterns.

You may also configure key grouping so that similar keys are grouped as a single entry. For example, you may wish to remove unique IDs from keys caching the same type of information. Groups are configured using a regular expression to "find and replace" parts of the key. An example is included in the configuration file:

```
Recorders\CacheInteractions::class => [
    // ...
    'groups' => [
        // '/:\d+/' => '.*',
    ],
],
```

The first pattern that matches will be used. If no patterns match, then the key will be captured as-is.

Exceptions

The `Exceptions` recorder captures information about reportable exceptions occurring in your application for display on the `Exceptions` card.

You may optionally adjust the `sample rate` and ignored exceptions patterns. You may also configure whether to capture the location that the exception originated from. The captured location will be displayed on the Pulse dashboard which can help to track down the exception origin; however, if the same exception occurs in multiple locations then it will appear multiple times for each unique location.

Queues

The `Queues` recorder captures information about your applications queues for display on the `Queues`.

You may optionally adjust the `sample rate` and ignored jobs patterns.

Slow Jobs

The `SlowJobs` recorder captures information about slow jobs occurring in your application for display on the `Slow Jobs` card.

You may optionally adjust the slow job threshold, `sample rate`, and ignored job patterns.

Slow Outgoing Requests

The `SlowOutgoingRequests` recorder captures information about outgoing HTTP requests made using Laravel's `HTTP client` that exceed the configured threshold for display on the `Slow Outgoing Requests` card.

You may optionally adjust the slow outgoing request threshold, `sample rate`, and ignored URL patterns.

You may also configure URL grouping so that similar URLs are grouped as a single entry. For example, you may wish to remove unique IDs from URL paths or group by domain only. Groups are configured using a regular expression to "find and replace" parts of the URL. Some examples are included in the configuration file:

```
Recorders\OutgoingRequests::class => [
    // ...
    'groups' => [
        // '#^https://api\.github\.com/repos/.*$#' => 'api.github.com/repos/*',
        // '#^https://([^\/]*)\.*$#' => '\1',
        // '#/\d+#' => '/*',
    ],
],
],
```

The first pattern that matches will be used. If no patterns match, then the URL will be captured as-is.

Slow Queries

The `SlowQueries` recorder captures any database queries in your application that exceed the configured threshold for display on the [Slow Queries](#) card.

You may optionally adjust the slow query threshold, [sample rate](#), and ignored query patterns. You may also configure whether to capture the query location. The captured location will be displayed on the Pulse dashboard which can help to track down the query origin; however, if the same query is made in multiple locations then it will appear multiple times for each unique location.

Slow Requests

The `Requests` recorder captures information about requests made to your application for display on the [Slow Requests](#) and [Application Usage](#) cards.

You may optionally adjust the slow route threshold, [sample rate](#), and ignored paths.

Servers

The `Servers` recorder captures CPU, memory, and storage usage of the servers that power your application for display on the [Servers](#) card. This recorder requires the `pulse:check` command to be running on each of the servers you wish to monitor.

Each reporting server must have a unique name. By default, Pulse will use the value returned by PHP's `gethostname` function. If you wish to customize this, you may set the `PULSE_SERVER_NAME` environment variable:

```
PULSE_SERVER_NAME=load-balancer
```

The Pulse configuration file also allows you to customize the directories that are monitored.

User Jobs

The `UserJobs` recorder captures information about the users dispatching jobs in your application for display on the [Application Usage](#) card.

You may optionally adjust the [sample rate](#) and ignored job patterns.

User Requests

The `UserRequests` recorder captures information about the users making requests to your application for display on the [Application Usage](#) card.

You may optionally adjust the [sample rate](#) and ignored job patterns.

Filtering

As we have seen, many [recorders](#) offer the ability to, via configuration, "ignore" incoming entries based on their value, such as a request's URL. But, sometimes it may be useful to filter out records based on other factors, such as the currently

authenticated user. To filter out these records, you may pass a closure to Pulse's `filter` method. Typically, the `filter` method should be invoked within the `boot` method of your application's `AppServiceProvider`:

```
use Illuminate\Support\Facades\Auth;
use Laravel\Pulse\Entry;
use Laravel\Pulse\Facades\Pulse;
use Laravel\Pulse\Value;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Pulse::filter(function (Entry|Value $entry) {
        return Auth::user()->isNotAdmin();
    });

    // ...
}
```

Performance

Pulse has been designed to drop into an existing application without requiring any additional infrastructure. However, for high-traffic applications, there are several ways of removing any impact Pulse may have on your application's performance.

Using a Different Database

For high-traffic applications, you may prefer to use a dedicated database connection for Pulse to avoid impacting your application database.

You may customize the [database connection](#) used by Pulse by setting the `PULSE_DB_CONNECTION` environment variable.

```
PULSE_DB_CONNECTION=pulse
```

Redis Ingest



The Redis Ingest requires Redis 6.2 or greater and `phpredis` or `predis` as the application's configured Redis client driver.

By default, Pulse will store entries directly to the [configured database connection](#) after the HTTP response has been sent to the client or a job has been processed; however, you may use Pulse's Redis ingest driver to send entries to a Redis stream instead. This can be enabled by configuring the `PULSE_INGEST_DRIVER` environment variable:

```
PULSE_INGEST_DRIVER=redis
```

Pulse will use your default [Redis connection](#) by default, but you may customize this via the `PULSE_REDIS_CONNECTION` environment variable:

```
PULSE_REDIS_CONNECTION=pulse
```

When using the Redis ingest, you will need to run the `pulse:work` command to monitor the stream and move entries from Redis into Pulse's database tables.

```
php artisan pulse:work
```



To keep the `pulse:work` process running permanently in the background, you should use a process monitor such as Supervisor to ensure that the Pulse worker does not stop running.

As the `pulse:work` command is a long-lived process, it will not see changes to your codebase without being restarted. You should gracefully restart the command by calling the `pulse:restart` command during your application's deployment process:

```
php artisan pulse:restart
```



Pulse uses the `cache` to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Sampling

By default, Pulse will capture every relevant event that occurs in your application. For high-traffic applications, this can result in needing to aggregate millions of database rows in the dashboard, especially for longer time periods.

You may instead choose to enable "sampling" on certain Pulse data recorders. For example, setting the sample rate to `0.1` on the `User Requests` recorder will mean that you only record approximately 10% of the requests to your application. In the dashboard, the values will be scaled up and prefixed with a `~` to indicate that they are an approximation.

In general, the more entries you have for a particular metric, the lower you can safely set the sample rate without sacrificing too much accuracy.

Trimming

Pulse will automatically trim its stored entries once they are outside of the dashboard window. Trimming occurs when ingesting data using a lottery system which may be customized in the Pulse [configuration file](#).

Handling Pulse Exceptions

If an exception occurs while capturing Pulse data, such as being unable to connect to the storage database, Pulse will silently fail to avoid impacting your application.

If you wish to customize how these exceptions are handled, you may provide a closure to the `handleExceptionsUsing` method:

```
use Laravel\Pulse\Facades\Pulse;
use Illuminate\Support\Facades\Log;

Pulse::handleExceptionsUsing(function ($e) {
    Log::debug('An exception happened in Pulse', [
        'message' => $e->getMessage(),
        'stack' => $e->getTraceAsString(),
    ]);
});
```

Custom Cards

Pulse allows you to build custom cards to display data relevant to your application's specific needs. Pulse uses [Livewire](#), so you may want to [review its documentation](#) before building your first custom card.

Card Components

Creating a custom card in Laravel Pulse starts with extending the base `Card` Livewire component and defining a corresponding view:

```
namespace App\Livewire\Pulse;

use Laravel\Pulse\Livewire\Card;
use Livewire\Attributes\Lazy;

#[Lazy]
class TopSellers extends Card
{
    public function render()
    {
        return view('livewire.pulse.top-sellers');
    }
}
```

When using Livewire's [lazy loading](#) feature, The `Card` component will automatically provide a placeholder that respects the `cols` and `rows` attributes passed to your component.

When writing your Pulse card's corresponding view, you may leverage Pulse's Blade components for a consistent look and feel:

```
<x-pulse::card :cols="$cols" :rows="$rows" :class="$class" wire:poll.5s="">
    <x-pulse::card-header name="Top Sellers">
        <x-slot:icon>
            ...
        </x-slot:icon>
    </x-pulse::card-header>

    <x-pulse::scroll :expand="$expand">
        ...
    </x-pulse::scroll>
</x-pulse::card>
```

The `$cols`, `$rows`, `$class`, and `$expand` variables should be passed to their respective Blade components so the card layout may be customized from the dashboard view. You may also wish to include the `wire:poll.5s=""` attribute in

your view to have the card automatically update.

Once you have defined your Livewire component and template, the card may be included in your [dashboard view](#):

```
<x-pulse>
  ...
<livewire:pulse.top-sellers cols="4" />
</x-pulse>
```



If your card is included in a package, you will need to register the component with Livewire using the `Livewire::component` method.

Styling

If your card requires additional styling beyond the classes and components included with Pulse, there are a few options for including custom CSS for your cards.

Laravel Vite Integration

If your custom card lives within your application's code base and you are using Laravel's [Vite integration](#), you may update your `vite.config.js` file to include a dedicated CSS entry point for your card:

```
laravel({
  input: [
    'resources/css/pulse/top-sellers.css',
    // ...
  ],
})
```

You may then use the `@vite` Blade directive in your [dashboard view](#), specifying the CSS entrypoint for your card:

```
<x-pulse>
  @vite('resources/css/pulse/top-sellers.css')

  ...
</x-pulse>
```

CSS Files

For other use cases, including Pulse cards contained within a package, you may instruct Pulse to load additional stylesheets by defining a `css` method on your Livewire component that returns the file path to your CSS file:

```
class TopSellers extends Card
{
    // ...

    protected function css()
    {
        return __DIR__.'/../../dist/top-sellers.css';
    }
}
```

When this card is included on the dashboard, Pulse will automatically include the contents of this file within a `<style>` tag so it does not need to be published to the `public` directory.

Tailwind CSS

When using Tailwind CSS, you should create a dedicated Tailwind configuration file to avoid loading unnecessary CSS or conflicting with Pulse's Tailwind classes:

```
export default {
    darkMode: 'class',
    important: '#top-sellers',
    content: [
        './resources/views/livewire/pulse/top-sellers.blade.php',
    ],
    corePlugins: {
        preflight: false,
    },
};
```

You may then specify the configuration file in your CSS entrypoint:

```
@config "../../tailwind.top-sellers.config.js";
@tailwind base;
@tailwind components;
@tailwind utilities;
```

You will also need to include an `id` or `class` attribute in your card's view that matches the selector passed to Tailwind's `important` selector strategy:

```
<x-pulse::card id="top-sellers" :cols="$cols" :rows="$rows" class="$class">
    ...
</x-pulse::card>
```

Data Capture and Aggregation

Custom cards may fetch and display data from anywhere; however, you may wish to leverage Pulse's powerful and efficient data recording and aggregation system.

Capturing Entries

Pulse allows you to record "entries" using the `Pulse::record` method:

```
use Laravel\Pulse\Facades\Pulse;

Pulse::record('user_sale', $user->id, $sale->amount)
    ->sum()
    ->count();
```

The first argument provided to the `record` method is the `type` for the entry you are recording, while the second argument is the `key` that determines how the aggregated data should be grouped. For most aggregation methods you will also need to specify a `value` to be aggregated. In the example above, the value being aggregated is `$sale->amount`. You may then invoke one or more aggregation methods (such as `sum`) so that Pulse may capture pre-aggregated values into "buckets" for efficient retrieval later.

The available aggregation methods are:

- `avg`
- `count`
- `max`
- `min`
- `sum`



When building a card package that captures the currently authenticated user ID, you should use the `Pulse::resolveAuthenticatedUserId()` method, which respects any [user resolver customizations](#) made to the application.

Retrieving Aggregate Data

When extending Pulse's `Card` Livewire component, you may use the `aggregate` method to retrieve aggregated data for the period being viewed in the dashboard:

```
class TopSellers extends Card
{
    public function render()
    {
        return view('livewire.pulse.top-sellers', [
            'topSellers' => $this->aggregate('user_sale', ['sum', 'count']);
        ]);
    }
}
```

The `aggregate` method returns a collection of PHP `stdClass` objects. Each object will contain the `key` property captured earlier, along with keys for each of the requested aggregates:

```
@foreach ($topSellers as $seller)
    {{ $seller->key }}
    {{ $seller->sum }}
    {{ $seller->count }}
@endforeach
```

Pulse will primarily retrieve data from the pre-aggregated buckets; therefore, the specified aggregates must have been captured up-front using the `Pulse::record` method. The oldest bucket will typically fall partially outside the period, so

Pulse will aggregate the oldest entries to fill the gap and give an accurate value for the entire period, without needing to aggregate the entire period on each poll request.

You may also retrieve a total value for a given type by using the `aggregateTotal` method. For example, the following method would retrieve the total of all user sales instead of grouping them by user.

```
$total = $this->aggregateTotal('user_sale', 'sum');
```

Displaying Users

When working with aggregates that record a user ID as the key, you may resolve the keys to user records using the `Pulse::resolveUsers` method:

```
$aggregates = $this->aggregate('user_sale', ['sum', 'count']);

$users = Pulse::resolveUsers($aggregates->pluck('key'));

return view('livewire.pulse.top-sellers', [
    'sellers' => $aggregates->map(fn ($aggregate) => (object) [
        'user' => $users->find($aggregate->key),
        'sum' => $aggregate->sum,
        'count' => $aggregate->count,
    ])
]);
```

The `find` method returns an object containing `name`, `extra`, and `avatar` keys, which you may optionally pass directly to the `<x-pulse::user-card>` Blade component:

```
<x-pulse::user-card :user="{{ $seller->user }}" :stats="{{ $seller->sum }}" />
```

Custom Recorders

Package authors may wish to provide recorder classes to allow users to configure the capturing of data.

Recorders are registered in the `recorders` section of the application's `config/pulse.php` configuration file:

```
[  
    // ...  
    'recorders' => [  
        Acme\Recorders\Deployments::class => [  
            // ...  
        ],  
        // ...  
    ],  
]
```

Recorders may listen to events by specifying a `$listen` property. Pulse will automatically register the listeners and call the recorders `record` method:

```
<?php

namespace Acme\Recorders;

use Acme\Events\Deployment;
use Illuminate\Support\Facades\Config;
use Laravel\Pulse\Facades\Pulse;

class Deployments
{
    /**
     * The events to listen for.
     *
     * @var list<class-string>
     */
    public array $listen = [
        Deployment::class,
    ];

    /**
     * Record the deployment.
     */
    public function record(Deployment $event): void
    {
        $config = Config::get('pulse.recorders.'.static::class);

        Pulse::record(
            // ...
        );
    }
}
```

Laravel Reverb

- [Introduction](#)
- [Installation](#)
- [Configuration](#)
 - [Application Credentials](#)
 - [Allowed Origins](#)
 - [Additional Applications](#)
 - [SSL](#)
- [Running the Server](#)
 - [Debugging](#)
 - [Restarting](#)
- [Running Reverb in Production](#)
 - [Open Files](#)
 - [Event Loop](#)
 - [Web Server](#)
 - [Ports](#)
 - [Process Management](#)
 - [Scaling](#)

Introduction

[Laravel Reverb](#) brings blazing-fast and scalable real-time WebSocket communication directly to your Laravel application, and provides seamless integration with Laravel's existing suite of event broadcasting tools.

Installation



Laravel Reverb requires PHP 8.2+ and Laravel 10.47+.

You may use the Composer package manager to install Reverb into your Laravel project:

```
composer require laravel/reverb
```

Once the package is installed, you may run Reverb's installation command to publish the configuration, add Reverb's required environment variables, and enable event broadcasting in your application:

```
php artisan reverb:install
```

Configuration

The `reverb:install` command will automatically configure Reverb using a sensible set of default options. If you would like to make any configuration changes, you may do so by updating Reverb's environment variables or by updating the `config/reverb.php` configuration file.

Application Credentials

In order to establish a connection to Reverb, a set of Reverb "application" credentials must be exchanged between the client and server. These credentials are configured on the server and are used to verify the request from the client. You

may define these credentials using the following environment variables:

```
REVERB_APP_ID=my-app-id
REVERB_APP_KEY=my-app-key
REVERB_APP_SECRET=my-app-secret
```

Allowed Origins

You may also define the origins from which client requests may originate by updating the value of the `allowed_origins` configuration value within the `apps` section of the `config/reverb.php` configuration file. Any requests from an origin not listed in your allowed origins will be rejected. You may allow all origins using `*`:

```
'apps' => [
    [
        'id' => 'my-app-id',
        'allowed_origins' => ['laravel.com'],
        // ...
    ]
]
```

Additional Applications

Typically, Reverb provides a WebSocket server for the application in which it is installed. However, it is possible to serve more than one application using a single Reverb installation.

For example, you may wish to maintain a single Laravel application which, via Reverb, provides WebSocket connectivity for multiple applications. This can be achieved by defining multiple `apps` in your application's `config/reverb.php` configuration file:

```
'apps' => [
    [
        'app_id' => 'my-app-one',
        // ...
    ],
    [
        'app_id' => 'my-app-two',
        // ...
    ],
]
```

SSL

In most cases, secure WebSocket connections are likely handled by an upstream web server (Nginx, etc.) before the request is proxied to your Reverb server.

However, it can sometimes be useful, such as during local development, for the Reverb server to handle secure connections directly. If you are using [Laravel Herd's](#) secure site functionality, or you are using [Laravel Valet](#) and have run the [secure command](#) against your application, you may use the Herd / Valet certificate generated for your site to secure your Reverb connections. To do so, set the `REVERB_HOST` environment variable to your site's hostname or explicitly pass the hostname option when starting the Reverb server:

```
php artisan reverb:start --host="0.0.0.0" --port=8080 --hostname="laravel.test"
```

Since Herd and Valet domains resolve to `localhost`, running the command above will result in your Reverb server being accessible via the secure WebSocket protocol (wss) at `wss://laravel.test:8080`.

You may also manually choose a certificate by defining `tls` options in your application's `config/reverb.php` configuration file. Within the array of `tls` options, you may provide any of the options supported by [PHP's SSL context options](#):

```
'options' => [
    'tls' => [
        'local_cert' => '/path/to/cert.pem'
    ],
],
```

Running the Server

The Reverb server can be started using the `reverb:start` Artisan command:

```
php artisan reverb:start
```

By default, the Reverb server will be started at `0.0.0.0:8080`, making it accessible from all network interfaces.

If you need to specify a custom host or port, you may do so via the `--host` and `--port` options when starting the server:

```
php artisan reverb:start --host=127.0.0.1 --port=9000
```

Alternatively, you may define `REVERB_SERVER_HOST` and `REVERB_SERVER_PORT` environment variables in your application's `.env` configuration file.

The `REVERB_SERVER_HOST` and `REVERB_SERVER_PORT` environment variables should not be confused with `REVERB_HOST` and `REVERB_PORT`. The former specify the host and port on which to run the Reverb server itself, while the latter pair instruct Laravel where to send broadcast messages. For example, in a production environment, you may route requests from your public Reverb hostname on port `443` to a Reverb server operating on `0.0.0.0:8080`. In this scenario, your environment variables would be defined as follows:

```
REVERB_SERVER_HOST=0.0.0.0
REVERB_SERVER_PORT=8080

REVERB_HOST=ws.laravel.com
REVERB_PORT=443
```

Debugging

To improve performance, Reverb does not output any debug information by default. If you would like to see the stream of data passing through your Reverb server, you may provide the `--debug` option to the `reverb:start` command:

```
php artisan reverb:start --debug
```

Restarting

Since Reverb is a long-running process, changes to your code will not be reflected without restarting the server via the `reverb:restart` Artisan command.

The `reverb:restart` command ensures all connections are gracefully terminated before stopping the server. If you are running Reverb with a process manager such as Supervisor, the server will be automatically restarted by the process manager after all connections have been terminated:

```
php artisan reverb:restart
```

Running Reverb in Production

Due to the long-running nature of WebSocket servers, you may need to make some optimizations to your server and hosting environment to ensure your Reverb server can effectively handle the optimal number of connections for the resources available on your server.



If your site is managed by [Laravel Forge](#), you may automatically optimize your server for Reverb directly from the "Application" panel. By enabling the Reverb integration, Forge will ensure your server is production-ready, including installing any required extensions and increasing the allowed number of connections.

Open Files

Each WebSocket connection is held in memory until either the client or server disconnects. In Unix and Unix-like environments, each connection is represented by a file. However, there are often limits on the number of allowed open files at both the operating system and application level.

Operating System

On a Unix based operating system, you may determine the allowed number of open files using the `ulimit` command:

```
ulimit -n
```

This command will display the open file limits allowed for different users. You may update these values by editing the `/etc/security/limits.conf` file. For example, updating the maximum number of open files to 10,000 for the `forge` user would look like the following:

```
# /etc/security/limits.conf
forge      soft  nofile  10000
forge      hard  nofile  10000
```

Event Loop

Under the hood, Reverb uses a ReactPHP event loop to manage WebSocket connections on the server. By default, this event loop is powered by `stream_select`, which doesn't require any additional extensions. However, `stream_select` is typically limited to 1,024 open files. As such, if you plan to handle more than 1,000 concurrent connections, you will need to use an alternative event loop not bound by the same restrictions.

Reverb will automatically switch to an `ext-event`, `ext-ev`, or `ext-uv` powered loop when available. All of these PHP extensions are available for install via PECL:

```
pecl install event
# or
pecl install ev
# or
pecl install uv
```

Web Server

In most cases, Reverb runs on a non web-facing port on your server. So, in order to route traffic to Reverb, you should configure a reverse proxy. Assuming Reverb is running on host `0.0.0.0` and port `8080` and your server utilizes the Nginx web server, a reverse proxy can be defined for your Reverb server using the following Nginx site configuration:

```
server {
    ...

    location / {
        proxy_http_version 1.1;
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header SERVER_PORT $server_port;
        proxy_set_header REMOTE_ADDR $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";

        proxy_pass http://0.0.0.0:8080;
    }

    ...
}
```

Typically, web servers are configured to limit the number of allowed connections in order to prevent overloading the server. To increase the number of allowed connections on an Nginx web server to 10,000, the `worker_rlimit_nofile` and `worker_connections` values of the `nginx.conf` file should be updated:

```
user forge;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;
worker_rlimit_nofile 10000;

events {
    worker_connections 10000;
    multi_accept on;
}
```

The configuration above will allow up to 10,000 Nginx workers per process to be spawned. In addition, this configuration sets Nginx's open file limit to 10,000.

Ports

Unix-based operating systems typically limit the number of ports which can be opened on the server. You may see the current allowed range via the following command:

```
cat /proc/sys/net/ipv4/ip_local_port_range
# 32768 60999
```

The output above shows the server can handle a maximum of 28,231 (60,999 - 32,768) connections since each connection requires a free port. Although we recommend [horizontal scaling](#) to increase the number of allowed connections, you may increase the number of available open ports by updating the allowed port range in your server's `/etc/sysctl.conf` configuration file.

Process Management

In most cases, you should use a process manager such as Supervisor to ensure the Reverb server is continually running. If you are using Supervisor to run Reverb, you should update the `minfds` setting of your server's `supervisor.conf` file to ensure Supervisor is able to open the files required to handle connections to your Reverb server:

```
[supervisord]
...
minfds=10000
```

Scaling

If you need to handle more connections than a single server will allow, you may scale your Reverb server horizontally. Utilizing the publish / subscribe capabilities of Redis, Reverb is able to manage connections across multiple servers. When a message is received by one of your application's Reverb servers, the server will use Redis to publish the incoming message to all other servers.

To enable horizontal scaling, you should set the `REVERB_SCALING_ENABLED` environment variable to `true` in your application's `.env` configuration file:

```
REVERB_SCALING_ENABLED=true
```

Next, you should have a dedicated, central Redis server to which all of the Reverb servers will communicate. Reverb will use the [default Redis connection configured for your application](#) to publish messages to all of your Reverb servers.

Once you have enabled Reverb's scaling option and configured a Redis server, you may simply invoke the `reverb:start` command on multiple servers that are able to communicate with your Redis server. These Reverb servers should be placed behind a load balancer that distributes incoming requests evenly among the servers.

Laravel Sail

- [Introduction](#)
- [Installation and Setup](#)
 - [Installing Sail Into Existing Applications](#)
 - [Configuring A Shell Alias](#)
- [Starting and Stopping Sail](#)
- [Executing Commands](#)
 - [Executing PHP Commands](#)
 - [Executing Composer Commands](#)
 - [Executing Artisan Commands](#)
 - [Executing Node / NPM Commands](#)
- [Interacting With Databases](#)
 - [MySQL](#)
 - [Redis](#)
 - [Meilisearch](#)
 - [Typesense](#)
- [File Storage](#)
- [Running Tests](#)
 - [Laravel Dusk](#)
- [Previewing Emails](#)
- [Container CLI](#)
- [PHP Versions](#)
- [Node Versions](#)
- [Sharing Your Site](#)
- [Debugging With Xdebug](#)
 - [Xdebug CLI Usage](#)
 - [Xdebug Browser Usage](#)
- [Customization](#)

Introduction

[Laravel Sail](#) is a light-weight command-line interface for interacting with Laravel's default Docker development environment. Sail provides a great starting point for building a Laravel application using PHP, MySQL, and Redis without requiring prior Docker experience.

At its heart, Sail is the `docker-compose.yml` file and the `sail` script that is stored at the root of your project. The `sail` script provides a CLI with convenient methods for interacting with the Docker containers defined by the `docker-compose.yml` file.

Laravel Sail is supported on macOS, Linux, and Windows (via [WSL2](#)).

Installation and Setup

Laravel Sail is automatically installed with all new Laravel applications so you may start using it immediately. To learn how to create a new Laravel application, please consult Laravel's [installation documentation](#) for your operating system. During installation, you will be asked to choose which Sail supported services your application will be interacting with.

Installing Sail Into Existing Applications

If you are interested in using Sail with an existing Laravel application, you may simply install Sail using the Composer package manager. Of course, these steps assume that your existing local development environment allows you to install Composer dependencies:

```
composer require laravel/sail --dev
```

After Sail has been installed, you may run the `sail:install` Artisan command. This command will publish Sail's `docker-compose.yml` file to the root of your application and modify your `.env` file with the required environment variables in order to connect to the Docker services:

```
php artisan sail:install
```

Finally, you may start Sail. To continue learning how to use Sail, please continue reading the remainder of this documentation:

```
./vendor/bin/sail up
```



If you are using Docker Desktop for Linux, you should use the `default` Docker context by executing the following command: `docker context use default`.

Adding Additional Services

If you would like to add an additional service to your existing Sail installation, you may run the `sail:add` Artisan command:

```
php artisan sail:add
```

Using Devcontainers

If you would like to develop within a [Devcontainer](#), you may provide the `--devcontainer` option to the `sail:install` command. The `--devcontainer` option will instruct the `sail:install` command to publish a default `.devcontainer/devcontainer.json` file to the root of your application:

```
php artisan sail:install --devcontainer
```

Configuring A Shell Alias

By default, Sail commands are invoked using the `vendor/bin/sail` script that is included with all new Laravel applications:

```
./vendor/bin/sail up
```

However, instead of repeatedly typing `vendor/bin/sail` to execute Sail commands, you may wish to configure a shell alias that allows you to execute Sail's commands more easily:

```
alias sail='sh $([ -f sail ] && echo sail || echo vendor/bin/sail)'
```

To make sure this is always available, you may add this to your shell configuration file in your home directory, such as `~/.zshrc` or `~/.bashrc`, and then restart your shell.

Once the shell alias has been configured, you may execute Sail commands by simply typing `sail`. The remainder of this documentation's examples will assume that you have configured this alias:

```
sail up
```

Starting and Stopping Sail

Laravel Sail's `docker-compose.yml` file defines a variety of Docker containers that work together to help you build Laravel applications. Each of these containers is an entry within the `services` configuration of your `docker-compose.yml` file. The `laravel.test` container is the primary application container that will be serving your application.

Before starting Sail, you should ensure that no other web servers or databases are running on your local computer. To start all of the Docker containers defined in your application's `docker-compose.yml` file, you should execute the `up` command:

```
sail up
```

To start all of the Docker containers in the background, you may start Sail in "detached" mode:

```
sail up -d
```

Once the application's containers have been started, you may access the project in your web browser at: <http://localhost>.

To stop all of the containers, you may simply press Control + C to stop the container's execution. Or, if the containers are running in the background, you may use the `stop` command:

```
sail stop
```

Executing Commands

When using Laravel Sail, your application is executing within a Docker container and is isolated from your local computer. However, Sail provides a convenient way to run various commands against your application such as arbitrary PHP commands, Artisan commands, Composer commands, and Node / NPM commands.

When reading the Laravel documentation, you will often see references to Composer, Artisan, and Node / NPM commands that do not reference Sail. Those examples assume that these tools are installed on your local computer. If you are using Sail for your local Laravel development environment, you should execute those commands using Sail:

```
# Running Artisan commands locally...
php artisan queue:work

# Running Artisan commands within Laravel Sail...
sail artisan queue:work
```

Executing PHP Commands

PHP commands may be executed using the `php` command. Of course, these commands will execute using the PHP version that is configured for your application. To learn more about the PHP versions available to Laravel Sail, consult the [PHP version documentation](#):

```
sail php --version

sail php script.php
```

Executing Composer Commands

Composer commands may be executed using the `composer` command. Laravel Sail's application container includes a Composer 2.x installation:

```
sail composer require laravel/sanctum
```

Installing Composer Dependencies for Existing Applications

If you are developing an application with a team, you may not be the one that initially creates the Laravel application. Therefore, none of the application's Composer dependencies, including Sail, will be installed after you clone the application's repository to your local computer.

You may install the application's dependencies by navigating to the application's directory and executing the following command. This command uses a small Docker container containing PHP and Composer to install the application's dependencies:

```
docker run --rm \
-u "$(id -u):$(id -g)" \
-v "$(pwd):/var/www/html" \
-w /var/www/html \
laravelsail/php83-composer:latest \
composer install --ignore-platform-reqs
```

When using the `laravelsail/phpXX-composer` image, you should use the same version of PHP that you plan to use for your application ([80](#), [81](#), [82](#), or [83](#)).

Executing Artisan Commands

Laravel Artisan commands may be executed using the `artisan` command:

```
sail artisan queue:work
```

Executing Node / NPM Commands

Node commands may be executed using the `node` command while NPM commands may be executed using the `npm` command:

```
sail node --version

sail npm run dev
```

If you wish, you may use Yarn instead of NPM:

```
sail yarn
```

Interacting With Databases

MySQL

As you may have noticed, your application's `docker-compose.yml` file contains an entry for a MySQL container. This container uses a [Docker volume](#) so that the data stored in your database is persisted even when stopping and restarting your containers.

In addition, the first time the MySQL container starts, it will create two databases for you. The first database is named using the value of your `DB_DATABASE` environment variable and is for your local development. The second is a dedicated

testing database named `testing` and will ensure that your tests do not interfere with your development data.

Once you have started your containers, you may connect to the MySQL instance within your application by setting your `DB_HOST` environment variable within your application's `.env` file to `mysql`.

To connect to your application's MySQL database from your local machine, you may use a graphical database management application such as [TablePlus](#). By default, the MySQL database is accessible at `localhost` port 3306 and the access credentials correspond to the values of your `DB_USERNAME` and `DB_PASSWORD` environment variables. Or, you may connect as the `root` user, which also utilizes the value of your `DB_PASSWORD` environment variable as its password.

Redis

Your application's `docker-compose.yml` file also contains an entry for a [Redis](#) container. This container uses a [Docker volume](#) so that the data stored in your Redis data is persisted even when stopping and restarting your containers. Once you have started your containers, you may connect to the Redis instance within your application by setting your `REDIS_HOST` environment variable within your application's `.env` file to `redis`.

To connect to your application's Redis database from your local machine, you may use a graphical database management application such as [TablePlus](#). By default, the Redis database is accessible at `localhost` port 6379.

Meilisearch

If you chose to install the [Meilisearch](#) service when installing Sail, your application's `docker-compose.yml` file will contain an entry for this powerful search-engine that is [compatible](#) with [Laravel Scout](#). Once you have started your containers, you may connect to the Meilisearch instance within your application by setting your `MEILISEARCH_HOST` environment variable to `http://meilisearch:7700`.

From your local machine, you may access Meilisearch's web based administration panel by navigating to `http://localhost:7700` in your web browser.

Typesense

If you chose to install the [Typesense](#) service when installing Sail, your application's `docker-compose.yml` file will contain an entry for this lightning fast, open-source search-engine that is natively integrated with [Laravel Scout](#). Once you have started your containers, you may connect to the Typesense instance within your application by setting the following environment variables:

```
TYPESENSE_HOST=typesense
TYPESENSE_PORT=8108
TYPESENSE_PROTOCOL=http
TYPESENSE_API_KEY=xyz
```

From your local machine, you may access Typesense's API via `http://localhost:8108`.

File Storage

If you plan to use Amazon S3 to store files while running your application in its production environment, you may wish to install the [MinIO](#) service when installing Sail. MinIO provides an S3 compatible API that you may use to develop locally using Laravel's `s3` file storage driver without creating "test" storage buckets in your production S3 environment. If you choose to install MinIO while installing Sail, a MinIO configuration section will be added to your application's `docker-compose.yml` file.

By default, your application's `filesystems` configuration file already contains a disk configuration for the `s3` disk. In addition to using this disk to interact with Amazon S3, you may use it to interact with any S3 compatible file storage service such as MinIO by simply modifying the associated environment variables that control its configuration. For example, when using MinIO, your filesystem environment variable configuration should be defined as follows:

```
FILESYSTEM_DISK=s3
AWS_ACCESS_KEY_ID=sail
AWS_SECRET_ACCESS_KEY=password
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=local
AWS_ENDPOINT=http://minio:9000
AWS_USE_PATH_STYLE_ENDPOINT=true
```

In order for Laravel's Flysystem integration to generate proper URLs when using MinIO, you should define the `AWS_URL` environment variable so that it matches your application's local URL and includes the bucket name in the URL path:

```
AWS_URL=http://localhost:9000/local
```

You may create buckets via the MinIO console, which is available at `http://localhost:8900`. The default username for the MinIO console is `sail` while the default password is `password`.



Generating temporary storage URLs via the `temporaryUrl` method is not supported when using MinIO.

Running Tests

Laravel provides amazing testing support out of the box, and you may use Sail's `test` command to run your applications [feature and unit tests](#). Any CLI options that are accepted by PHPUnit may also be passed to the `test` command:

```
sail test
sail test --group orders
```

The Sail `test` command is equivalent to running the `test` Artisan command:

```
sail artisan test
```

By default, Sail will create a dedicated `testing` database so that your tests do not interfere with the current state of your database. In a default Laravel installation, Sail will also configure your `phpunit.xml` file to use this database when executing your tests:

```
<env name="DB_DATABASE" value="testing"/>
```

Laravel Dusk

[Laravel Dusk](#) provides an expressive, easy-to-use browser automation and testing API. Thanks to Sail, you may run these tests without ever installing Selenium or other tools on your local computer. To get started, uncomment the Selenium service in your application's `docker-compose.yml` file:

```
selenium:
    image: 'selenium/standalone-chrome'
    extra_hosts:
        - 'host.docker.internal:host-gateway'
    volumes:
        - '/dev/shm:/dev/shm'
    networks:
        - sail
```

Next, ensure that the `laravel.test` service in your application's `docker-compose.yml` file has a `depends_on` entry for `selenium`:

```
depends_on:
    - mysql
    - redis
    - selenium
```

Finally, you may run your Dusk test suite by starting Sail and running the `dusk` command:

```
sail dusk
```

Selenium on Apple Silicon

If your local machine contains an Apple Silicon chip, your `selenium` service must use the `seleniarm/standalone-chromium` image:

```
selenium:
    image: 'seleniarm/standalone-chromium'
    extra_hosts:
        - 'host.docker.internal:host-gateway'
    volumes:
        - '/dev/shm:/dev/shm'
    networks:
        - sail
```

Previewing Emails

Laravel Sail's default `docker-compose.yml` file contains a service entry for [Mailpit](#). Mailpit intercepts emails sent by your application during local development and provides a convenient web interface so that you can preview your email messages in your browser. When using Sail, Mailpit's default host is `mailpit` and is available via port 1025:

```
MAIL_HOST=mailpit
MAIL_PORT=1025
MAIL_ENCRYPTION=null
```

When Sail is running, you may access the Mailpit web interface at: <http://localhost:8025>

Container CLI

Sometimes you may wish to start a Bash session within your application's container. You may use the `shell` command to connect to your application's container, allowing you to inspect its files and installed services as well execute arbitrary shell commands within the container:

```
sail shell
sail root-shell
```

To start a new [Laravel Tinker](#) session, you may execute the `tinker` command:

```
sail tinker
```

PHP Versions

Sail currently supports serving your application via PHP 8.3, 8.2, 8.1, or PHP 8.0. The default PHP version used by Sail is currently PHP 8.3. To change the PHP version that is used to serve your application, you should update the `build` definition of the `laravel.test` container in your application's `docker-compose.yml` file:

```
# PHP 8.3
context: ./vendor/laravel/sail/runtimes/8.3

# PHP 8.2
context: ./vendor/laravel/sail/runtimes/8.2

# PHP 8.1
context: ./vendor/laravel/sail/runtimes/8.1

# PHP 8.0
context: ./vendor/laravel/sail/runtimes/8.0
```

In addition, you may wish to update your `image` name to reflect the version of PHP being used by your application. This option is also defined in your application's `docker-compose.yml` file:

```
image: sail-8.1/app
```

After updating your application's `docker-compose.yml` file, you should rebuild your container images:

```
sail build --no-cache
sail up
```

Node Versions

Sail installs Node 20 by default. To change the Node version that is installed when building your images, you may update the `build.args` definition of the `laravel.test` service in your application's `docker-compose.yml` file:

```
build:
  args:
    WWWGROUP: '${WWWGROUP}'
    NODE_VERSION: '18'
```

After updating your application's `docker-compose.yml` file, you should rebuild your container images:

```
sail build --no-cache
sail up
```

Sharing Your Site

Sometimes you may need to share your site publicly in order to preview your site for a colleague or to test webhook integrations with your application. To share your site, you may use the `share` command. After executing this command, you will be issued a random `laravel-sail.site` URL that you may use to access your application:

```
sail share
```

When sharing your site via the `share` command, you should configure your application's trusted proxies within the `TrustProxies` middleware. Otherwise, URL generation helpers such as `url` and `route` will be unable to determine the correct HTTP host that should be used during URL generation:

```
/**
 * The trusted proxies for this application.
 *
 * @var array|string|null
 */
protected $proxies = '*';
```

If you would like to choose the subdomain for your shared site, you may provide the `subdomain` option when executing the `share` command:

```
sail share --subdomain=my-sail-site
```



The `share` command is powered by [Expose](#), an open source tunneling service by [BeyondCode](#).

Debugging With Xdebug

Laravel Sail's Docker configuration includes support for [Xdebug](#), a popular and powerful debugger for PHP. In order to enable Xdebug, you will need to add a few variables to your application's `.env` file to [configure Xdebug](#). To enable Xdebug you must set the appropriate mode(s) before starting Sail:

```
SAIL_XDEBUG_MODE=develop,debug,coverage
```

Linux Host IP Configuration

Internally, the `XDEBUG_CONFIG` environment variable is defined as `client_host=host.docker.internal` so that Xdebug will be properly configured for Mac and Windows (WSL2). If your local machine is running Linux, you should ensure that you are running Docker Engine 17.06.0+ and Compose 1.16.0+. Otherwise, you will need to manually define this environment variable as shown below.

First, you should determine the correct host IP address to add to the environment variable by running the following command. Typically, the `<container-name>` should be the name of the container that serves your application and often ends with `_laravel.test_1`:

```
docker inspect -f {{range.NetworkSettings.Networks}}{{.Gateway}}{{end}} <container-name>
```

Once you have obtained the correct host IP address, you should define the `SAIL_XDEBUG_CONFIG` variable within your application's `.env` file:

```
SAIL_XDEBUG_CONFIG="client_host=<host-ip-address>"
```

Xdebug CLI Usage

A `sail debug` command may be used to start a debugging session when running an Artisan command:

```
# Run an Artisan command without Xdebug...
sail artisan migrate

# Run an Artisan command with Xdebug...
sail debug migrate
```

Xdebug Browser Usage

To debug your application while interacting with the application via a web browser, follow the [instructions provided by Xdebug](#) for initiating an Xdebug session from the web browser.

If you're using PhpStorm, please review JetBrains' documentation regarding [zero-configuration debugging](#).



Laravel Sail relies on `artisan serve` to serve your application. The `artisan serve` command only accepts the `XDEBUG_CONFIG` and `XDEBUG_MODE` variables as of Laravel version 8.53.0. Older versions of Laravel (8.52.0 and below) do not support these variables and will not accept debug connections.

Customization

Since Sail is just Docker, you are free to customize nearly everything about it. To publish Sail's own Dockerfiles, you may execute the `sail:publish` command:

```
sail artisan sail:publish
```

After running this command, the Dockerfiles and other configuration files used by Laravel Sail will be placed within a `docker` directory in your application's root directory. After customizing your Sail installation, you may wish to change the image name for the application container in your application's `docker-compose.yml` file. After doing so, rebuild your application's containers using the `build` command. Assigning a unique name to the application image is particularly important if you are using Sail to develop multiple Laravel applications on a single machine:

```
sail build --no-cache
```

Laravel Sanctum

- [Introduction](#)
 - [How it Works](#)
- [Installation](#)
- [Configuration](#)
 - [Overriding Default Models](#)
- [API Token Authentication](#)
 - [Issuing API Tokens](#)
 - [Token Abilities](#)
 - [Protecting Routes](#)
 - [Revoking Tokens](#)
 - [Token Expiration](#)
- [SPA Authentication](#)
 - [Configuration](#)
 - [Authenticating](#)
 - [Protecting Routes](#)
 - [Authorizing Private Broadcast Channels](#)
- [Mobile Application Authentication](#)
 - [Issuing API Tokens](#)
 - [Protecting Routes](#)
 - [Revoking Tokens](#)
- [Testing](#)

Introduction

[Laravel Sanctum](#) provides a featherweight authentication system for SPAs (single page applications), mobile applications, and simple, token based APIs. Sanctum allows each user of your application to generate multiple API tokens for their account. These tokens may be granted abilities / scopes which specify which actions the tokens are allowed to perform.

How it Works

Laravel Sanctum exists to solve two separate problems. Let's discuss each before digging deeper into the library.

API Tokens

First, Sanctum is a simple package you may use to issue API tokens to your users without the complication of OAuth. This feature is inspired by GitHub and other applications which issue "personal access tokens". For example, imagine the "account settings" of your application has a screen where a user may generate an API token for their account. You may use Sanctum to generate and manage those tokens. These tokens typically have a very long expiration time (years), but may be manually revoked by the user at anytime.

Laravel Sanctum offers this feature by storing user API tokens in a single database table and authenticating incoming HTTP requests via the `Authorization` header which should contain a valid API token.

SPA Authentication

Second, Sanctum exists to offer a simple way to authenticate single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository, such as a SPA created using Vue CLI or a Next.js application.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. Typically, Sanctum utilizes Laravel's `web` authentication guard to accomplish this. This provides

the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.

Sanctum will only attempt to authenticate using cookies when the incoming request originates from your own SPA frontend. When Sanctum examines an incoming HTTP request, it will first check for an authentication cookie and, if none is present, Sanctum will then examine the `Authorization` header for a valid API token.



It is perfectly fine to use Sanctum only for API token authentication or only for SPA authentication. Just because you use Sanctum does not mean you are required to use both features it offers.

Installation



The most recent versions of Laravel already include Laravel Sanctum. However, if your application's `composer.json` file does not include `laravel/sanctum`, you may follow the installation instructions below.

You may install Laravel Sanctum via the Composer package manager:

```
composer require laravel/sanctum
```

Next, you should publish the Sanctum configuration and migration files using the `vendor:publish` Artisan command. The `sanctum` configuration file will be placed in your application's `config` directory:

```
php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"
```

Finally, you should run your database migrations. Sanctum will create one database table in which to store API tokens:

```
php artisan migrate
```

Next, if you plan to utilize Sanctum to authenticate a SPA, you should add Sanctum's middleware to your `api` middleware group within your application's `app/Http/Kernel.php` file:

```
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

Migration Customization

If you are not going to use Sanctum's default migrations, you should call the `Sanctum::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations by executing the following command: `php artisan vendor:publish --tag=sanctum-migrations`

Configuration

Overriding Default Models

Although not typically required, you are free to extend the `PersonalAccessToken` model used internally by Sanctum:

```
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;

class PersonalAccessToken extends SanctumPersonalAccessToken
{
    // ...
}
```

Then, you may instruct Sanctum to use your custom model via the `usePersonalAccessTokenModel` method provided by Sanctum. Typically, you should call this method in the `boot` method of one of your application's service providers:

```
use App\Models\Sanctum\PersonalAccessToken;
use Laravel\Sanctum\Sanctum;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Sanctum::usePersonalAccessTokenModel(PersonalAccessToken::class);
}
```

API Token Authentication



You should not use API tokens to authenticate your own first-party SPA. Instead, use Sanctum's built-in [SPA authentication features](#).

Issuing API Tokens

Sanctum allows you to issue API tokens / personal access tokens that may be used to authenticate API requests to your application. When making requests using API tokens, the token should be included in the `Authorization` header as a `Bearer` token.

To begin issuing tokens for users, your User model should use the `Laravel\Sanctum\HasApiTokens` trait:

```
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

To issue a token, you may use the `createToken` method. The `createToken` method returns a `Laravel\Sanctum\NewAccessToken` instance. API tokens are hashed using SHA-256 hashing before being stored in your database, but you may access the plain-text value of the token using the `plainTextToken` property of the `NewAccessToken` instance. You should display this value to the user immediately after the token has been created:

```
use Illuminate\Http\Request;

Route::post('/tokens/create', function (Request $request) {
    $token = $request->user()->createToken($request->token_name);

    return ['token' => $token->plainTextToken];
});
```

You may access all of the user's tokens using the `tokens` Eloquent relationship provided by the `HasApiTokens` trait:

```
foreach ($user->tokens as $token) {
    // ...
}
```

Token Abilities

Sanctum allows you to assign "abilities" to tokens. Abilities serve a similar purpose as OAuth's "scopes". You may pass an array of string abilities as the second argument to the `createToken` method:

```
return $user->createToken('token-name', ['server:update'])->plainTextToken;
```

When handling an incoming request authenticated by Sanctum, you may determine if the token has a given ability using the `tokenCan` method:

```
if ($user->tokenCan('server:update')) {
    // ...
}
```

Token Ability Middleware

Sanctum also includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given ability. To get started, add the following middleware to the `$middlewareAliases` property of your application's `app/Http/Kernel.php` file:

```
'abilities' => \Laravel\Sanctum\Http\Middleware\CheckAbilities::class,
'ability' => \Laravel\Sanctum\Http\Middleware\CheckForAnyAbility::class,
```

The `abilities` middleware may be assigned to a route to verify that the incoming request's token has all of the listed abilities:

```
Route::get('/orders', function () {
    // Token has both "check-status" and "place-orders" abilities...
})->middleware(['auth:sanctum', 'abilities:check-status,place-orders']);
```

The `ability` middleware may be assigned to a route to verify that the incoming request's token has *at least one* of the listed abilities:

```
Route::get('/orders', function () {
    // Token has the "check-status" or "place-orders" ability...
})->middleware(['auth:sanctum', 'ability:check-status,place-orders']);
```

First-Party UI Initiated Requests

For convenience, the `tokenCan` method will always return `true` if the incoming authenticated request was from your first-party SPA and you are using Sanctum's built-in [SPA authentication](#).

However, this does not necessarily mean that your application has to allow the user to perform the action. Typically, your application's [authorization policies](#) will determine if the token has been granted the permission to perform the abilities as well as check that the user instance itself should be allowed to perform the action.

For example, if we imagine an application that manages servers, this might mean checking that token is authorized to update servers **and** that the server belongs to the user:

```
return $request->user()->id === $server->user_id &&
    $request->user()->tokenCan('server:update')
```

At first, allowing the `tokenCan` method to be called and always return `true` for first-party UI initiated requests may seem strange; however, it is convenient to be able to always assume an API token is available and can be inspected via the `tokenCan` method. By taking this approach, you may always call the `tokenCan` method within your application's authorizations policies without worrying about whether the request was triggered from your application's UI or was initiated by one of your API's third-party consumers.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your protected routes within your `routes/web.php` and `routes/api.php` route files. This guard will ensure that incoming requests are authenticated as either stateful, cookie authenticated requests or contain a valid API token header if the request is from a third party.

You may be wondering why we suggest that you authenticate the routes within your application's `routes/web.php` file using the `sanctum` guard. Remember, Sanctum will first attempt to authenticate incoming requests using Laravel's typical session authentication cookie. If that cookie is not present then Sanctum will attempt to authenticate the request using a token in the request's `Authorization` header. In addition, authenticating all requests using Sanctum ensures that we may always call the `tokenCan` method on the currently authenticated user instance:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Revoking Tokens

You may "revoke" tokens by deleting them from your database using the `tokens` relationship that is provided by the `Laravel\Sanctum\HasApiTokens` trait:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke the token that was used to authenticate the current request...
$request->user()->currentAccessToken()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

Token Expiration

By default, Sanctum tokens never expire and may only be invalidated by [revoking the token](#). However, if you would like to configure an expiration time for your application's API tokens, you may do so via the `expiration` configuration option defined in your application's `sanctum` configuration file. This configuration option defines the number of minutes until an issued token will be considered expired:

```
'expiration' => 525600,
```

If you would like to specify the expiration time of each token independently, you may do so by providing the expiration time as the third argument to the `createToken` method:

```
return $user->createToken(
    'token-name', ['*'], now()->addWeek()
)->plainTextToken;
```

If you have configured a token expiration time for your application, you may also wish to [schedule a task](#) to prune your application's expired tokens. Thankfully, Sanctum includes a `sanctum:prune-expired` Artisan command that you may use to accomplish this. For example, you may configure a scheduled tasks to delete all expired token database records that have been expired for at least 24 hours:

```
$schedule->command('sanctum:prune-expired --hours=24')->daily();
```

SPA Authentication

Sanctum also exists to provide a simple method of authenticating single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. This approach to authentication provides the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.



In order to authenticate, your SPA and API must share the same top-level domain. However, they may be placed on different subdomains. Additionally, you should ensure that you send the `Accept: application/json` header and either the `Referer` or `Origin` header with your request.

Configuration

Configuring Your First-Party Domains

First, you should configure which domains your SPA will be making requests from. You may configure these domains using the `stateful` configuration option in your `sanctum` configuration file. This configuration setting determines which domains will maintain "stateful" authentication using Laravel session cookies when making requests to your API.



If you are accessing your application via a URL that includes a port (`127.0.0.1:8000`), you should ensure that you include the port number with the domain.

Sanctum Middleware

Next, you should add Sanctum's middleware to your `api` middleware group within your `app/Http/Kernel.php` file. This middleware is responsible for ensuring that incoming requests from your SPA can authenticate using Laravel's session cookies, while still allowing requests from third parties or mobile applications to authenticate using API tokens:

```
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class.'api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

CORS and Cookies

If you are having trouble authenticating with your application from a SPA that executes on a separate subdomain, you have likely misconfigured your CORS (Cross-Origin Resource Sharing) or session cookie settings.

You should ensure that your application's CORS configuration is returning the `Access-Control-Allow-Credentials` header with a value of `True`. This may be accomplished by setting the `supports_credentials` option within your application's `config/cors.php` configuration file to `true`.

In addition, you should enable the `withCredentials` and `withXSRFToken` options on your application's global `axios` instance. Typically, this should be performed in your `resources/js/bootstrap.js` file. If you are not using Axios to make HTTP requests from your frontend, you should perform the equivalent configuration on your own HTTP client:

```
axios.defaults.withCredentials = true;
axios.defaults.withXSRFToken = true;
```

Finally, you should ensure your application's session cookie domain configuration supports any subdomain of your root domain. You may accomplish this by prefixing the domain with a leading `.` within your application's `config/session.php` configuration file:

```
'domain' => '.domain.com',
```

Authenticating

CSRF Protection

To authenticate your SPA, your SPA's "login" page should first make a request to the `/sanctum/csrf-cookie` endpoint to initialize CSRF protection for the application:

```
axios.get('/sanctum/csrf-cookie').then(response => {
    // Login...
});
```

During this request, Laravel will set an `XSRF-TOKEN` cookie containing the current CSRF token. This token should then be passed in an `X-XSRF-TOKEN` header on subsequent requests, which some HTTP client libraries like Axios and the Angular HttpClient will do automatically for you. If your JavaScript HTTP library does not set the value for you, you will need to manually set the `X-XSRF-TOKEN` header to match the value of the `XSRF-TOKEN` cookie that is set by this route.

Logging In

Once CSRF protection has been initialized, you should make a `POST` request to your Laravel application's `/login` route. This `/login` route may be [implemented manually](#) or using a headless authentication package like [Laravel Fortify](#).

If the login request is successful, you will be authenticated and subsequent requests to your application's routes will automatically be authenticated via the session cookie that the Laravel application issued to your client. In addition, since your application already made a request to the `/sanctum/csrf-cookie` route, subsequent requests should automatically

receive CSRF protection as long as your JavaScript HTTP client sends the value of the `XSRF-TOKEN` cookie in the `X-XSRF-TOKEN` header.

Of course, if your user's session expires due to lack of activity, subsequent requests to the Laravel application may receive 401 or 419 HTTP error response. In this case, you should redirect the user to your SPA's login page.



You are free to write your own `/login` endpoint; however, you should ensure that it authenticates the user using the standard, [session based authentication services that Laravel provides](#). Typically, this means using the `web` authentication guard.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your API routes within your `routes/api.php` file. This guard will ensure that incoming requests are authenticated as either a stateful authenticated requests from your SPA or contain a valid API token header if the request is from a third party:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Authorizing Private Broadcast Channels

If your SPA needs to authenticate with [private presence broadcast channels](#), you should place the `Broadcast::routes` method call within your `routes/api.php` file:

```
Broadcast::routes(['middleware' => ['auth:sanctum']]));
```

Next, in order for Pusher's authorization requests to succeed, you will need to provide a custom Pusher `authorizer` when initializing [Laravel Echo](#). This allows your application to configure Pusher to use the `axios` instance that is [properly configured for cross-domain requests](#):

```
window.Echo = new Echo({
    broadcaster: "pusher",
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    encrypted: true,
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(false, response.data);
                })
                .catch(error => {
                    callback(true, error);
                });
            }
        };
    },
})
```

Mobile Application Authentication

You may also use Sanctum tokens to authenticate your mobile application's requests to your API. The process for authenticating mobile application requests is similar to authenticating third-party API requests; however, there are small differences in how you will issue the API tokens.

Issuing API Tokens

To get started, create a route that accepts the user's email / username, password, and device name, then exchanges those credentials for a new Sanctum token. The "device name" given to this endpoint is for informational purposes and may be any value you wish. In general, the device name value should be a name the user would recognize, such as "Nuno's iPhone 12".

Typically, you will make a request to the token endpoint from your mobile application's "login" screen. The endpoint will return the plain-text API token which may then be stored on the mobile device and used to make additional API requests:

```

use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

Route::post('/sanctum/token', function (Request $request) {
    $request->validate([
        'email' => 'required|email',
        'password' => 'required',
        'device_name' => 'required',
    ]);

    $user = User::where('email', $request->email)->first();

    if (! $user || ! Hash::check($request->password, $user->password)) {
        throw ValidationException::withMessages([
            'email' => ['The provided credentials are incorrect.'],
        ]);
    }

    return $user->createToken($request->device_name)->plainTextToken;
});

```

When the mobile application uses the token to make an API request to your application, it should pass the token in the `Authorization` header as a `Bearer` token.



When issuing tokens for a mobile application, you are also free to specify [token abilities](#).

Protecting Routes

As previously documented, you may protect routes so that all incoming requests must be authenticated by attaching the `sanctum` authentication guard to the routes:

```

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});

```

Revoking Tokens

To allow users to revoke API tokens issued to mobile devices, you may list them by name, along with a "Revoke" button, within an "account settings" portion of your web application's UI. When the user clicks the "Revoke" button, you can delete the token from the database. Remember, you can access a user's API tokens via the `tokens` relationship provided by the `Laravel\Sanctum\HasApiTokens` trait:

```

// Revoke all tokens...
$user->tokens()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();

```

Testing

While testing, the `Sanctum::actingAs` method may be used to authenticate a user and specify which abilities should be granted to their token:

```
use App\Models\User;
use Laravel\Sanctum\Sanctum;

public function test_task_list_can_be_retrieved(): void
{
    Sanctum::actingAs(
        User::factory()->create(),
        ['view-tasks']
    );

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

    $response->assertOk();
}
```

If you would like to grant all abilities to the token, you should include `*` in the ability list provided to the `actingAs` method:

```
Sanctum::actingAs(
    User::factory()->create(),
    ['*']
);
```

Laravel Scout

- [Introduction](#)
- [Installation](#)
 - [Queueing](#)
- [Driver Prerequisites](#)
 - [Algolia](#)
 - [Meilisearch](#)
 - [Typesense](#)
- [Configuration](#)
 - [Configuring Model Indexes](#)
 - [Configuring Searchable Data](#)
 - [Configuring the Model ID](#)
 - [Configuring Search Engines per Model](#)
 - [Identifying Users](#)
- [Database / Collection Engines](#)
 - [Database Engine](#)
 - [Collection Engine](#)
- [Indexing](#)
 - [Batch Import](#)
 - [Adding Records](#)
 - [Updating Records](#)
 - [Removing Records](#)
 - [Pausing Indexing](#)
 - [Conditionally Searchable Model Instances](#)
- [Searching](#)
 - [Where Clauses](#)
 - [Pagination](#)
 - [Soft Deleting](#)
 - [Customizing Engine Searches](#)
- [Custom Engines](#)

Introduction

[Laravel Scout](#) provides a simple, driver based solution for adding full-text search to your [Eloquent models](#). Using model observers, Scout will automatically keep your search indexes in sync with your Eloquent records.

Currently, Scout ships with [Algolia](#), [Meilisearch](#), [Typesense](#), and MySQL / PostgreSQL (`database`) drivers. In addition, Scout includes a "collection" driver that is designed for local development usage and does not require any external dependencies or third-party services. Furthermore, writing custom drivers is simple and you are free to extend Scout with your own search implementations.

Installation

First, install Scout via the Composer package manager:

```
composer require laravel/scout
```

After installing Scout, you should publish the Scout configuration file using the `vendor:publish` Artisan command. This command will publish the `scout.php` configuration file to your application's `config` directory:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

Finally, add the `Laravel\Scout\Searchable` trait to the model you would like to make searchable. This trait will register a model observer that will automatically keep the model in sync with your search driver:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;
}
```

Queueing

While not strictly required to use Scout, you should strongly consider configuring a [queue driver](#) before using the library. Running a queue worker will allow Scout to queue all operations that sync your model information to your search indexes, providing much better response times for your application's web interface.

Once you have configured a queue driver, set the value of the `queue` option in your `config/scout.php` configuration file to `true`:

```
'queue' => true,
```

Even when the `queue` option is set to `false`, it's important to remember that some Scout drivers like Algolia and Meilisearch always index records asynchronously. Meaning, even though the index operation has completed within your Laravel application, the search engine itself may not reflect the new and updated records immediately.

To specify the connection and queue that your Scout jobs utilize, you may define the `queue` configuration option as an array:

```
'queue' => [
    'connection' => 'redis',
    'queue' => 'scout'
],
```

Of course, if you customize the connection and queue that Scout jobs utilize, you should run a queue worker to process jobs on that connection and queue:

```
php artisan queue:work redis --queue=scout
```

Driver Prerequisites

Algolia

When using the Algolia driver, you should configure your Algolia `id` and `secret` credentials in your `config/scout.php` configuration file. Once your credentials have been configured, you will also need to install the Algolia PHP SDK via the Composer package manager:

```
composer require algolia/algoliasearch-client-php
```

Meilisearch

[Meilisearch](#) is a blazingly fast and open source search engine. If you aren't sure how to install Meilisearch on your local machine, you may use [Laravel Sail](#), Laravel's officially supported Docker development environment.

When using the Meilisearch driver you will need to install the Meilisearch PHP SDK via the Composer package manager:

```
composer require meilisearch/meilisearch-php http-interop/http-factory-guzzle
```

Then, set the `SCOUT_DRIVER` environment variable as well as your Meilisearch `host` and `key` credentials within your application's `.env` file:

```
SCOUT_DRIVER=meilisearch
MEILISEARCH_HOST=http://127.0.0.1:7700
MEILISEARCH_KEY=masterKey
```

For more information regarding Meilisearch, please consult the [Meilisearch documentation](#).

In addition, you should ensure that you install a version of `meilisearch/meilisearch-php` that is compatible with your Meilisearch binary version by reviewing [Meilisearch's documentation regarding binary compatibility](#).



When upgrading Scout on an application that utilizes Meilisearch, you should always [review any additional breaking changes](#) to the Meilisearch service itself.

Typesense

[Typesense](#) is a lightning-fast, open source search engine and supports keyword search, semantic search, geo search, and vector search.

You can [self-host Typesense](#) or use [Typesense Cloud](#).

To get started using Typesense with Scout, install the Typesense PHP SDK via the Composer package manager:

```
composer require typesense/typesense-php
```

Then, set the `SCOUT_DRIVER` environment variable as well as your Typesense host and API key credentials within your application's `.env` file:

```
SCOUT_DRIVER=typesense
TYPESENSE_API_KEY=masterKey
TYPESENSE_HOST=localhost
```

If needed, you may also specify your installation's port, path, and protocol:

```
TYPESENSE_PORT=8108
TYPESENSE_PATH=
TYPESENSE_PROTOCOL=http
```

Additional settings and schema definitions for your Typesense collections can be found within your application's `config/scout.php` configuration file. For more information regarding Typesense, please consult the [Typesense documentation](#).

Preparing Data for Storage in Typesense

When utilizing Typesense, your searchable model's must define a `toSearchableArray` method that casts your model's primary key to a string and creation date to a UNIX timestamp:

```
/**
 * Get the indexable data array for the model.
 *
 * @return array<string, mixed>
 */
public function toSearchableArray()
{
    return array_merge($this->toArray(), [
        'id' => (string) $this->id,
        'created_at' => $this->created_at->timestamp,
    ]);
}
```

You should also define your Typesense collection schemas in your application's `config/scout.php` file. A collection schema describes the data types of each field that is searchable via Typesense. For more information on all available schema options, please consult the [Typesense documentation](#).

If you need to change your Typesense collection's schema after it has been defined, you may either run `scout:flush` and `scout:import`, which will delete all existing indexed data and recreate the schema. Or, you may use Typesense's API to modify the collection's schema without removing any indexed data.

If your searchable model is soft deletable, you should define a `__soft_deleted` field in the model's corresponding Typesense schema within your application's `config/scout.php` configuration file:

```
User::class => [
    'collection-schema' => [
        'fields' => [
            // ...
            [
                'name' => '__soft_deleted',
                'type' => 'int32',
                'optional' => true,
            ],
        ],
    ],
],
```

Dynamic Search Parameters

Typesense allows you to modify your [search parameters](#) dynamically when performing a search operation via the `options` method:

```
use App\Models\Todo;

Todo::search('Groceries')->options([
    'query_by' => 'title, description'
])->get();
```

Configuration

Configuring Model Indexes

Each Eloquent model is synced with a given search "index", which contains all of the searchable records for that model. In other words, you can think of each index like a MySQL table. By default, each model will be persisted to an index matching the model's typical "table" name. Typically, this is the plural form of the model name; however, you are free to customize the model's index by overriding the `searchableAs` method on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the name of the index associated with the model.
     */
    public function searchableAs(): string
    {
        return 'posts_index';
    }
}
```

Configuring Searchable Data

By default, the entire `toArray` form of a given model will be persisted to its search index. If you would like to customize the data that is synchronized to the search index, you may override the `toSearchableArray` method on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array<string, mixed>
     */
    public function toSearchableArray(): array
    {
        $array = $this->toArray();

        // Customize the data array...

        return $array;
    }
}
```

Some search engines such as Meilisearch will only perform filter operations (`>`, `<`, etc.) on data of the correct type. So, when using these search engines and customizing your searchable data, you should ensure that numeric values are cast to their correct type:

```
public function toSearchableArray()
{
    return [
        'id' => (int) $this->id,
        'name' => $this->name,
        'price' => (float) $this->price,
    ];
}
```

Configuring Filterable Data and Index Settings (Meilisearch)

Unlike Scout's other drivers, Meilisearch requires you to pre-define index search settings such as filterable attributes, sortable attributes, and [other supported settings fields](#).

Filterable attributes are any attributes you plan to filter on when invoking Scout's `where` method, while sortable attributes are any attributes you plan to sort by when invoking Scout's `orderBy` method. To define your index settings, adjust the `index-settings` portion of your `meilisearch` configuration entry in your application's `scout` configuration file:

```

use App\Models\User;
use App\Models\Flight;

'meilisearch' => [
    'host' => env('MEILISEARCH_HOST', 'http://localhost:7700'),
    'key' => env('MEILISEARCH_KEY', null),
    'index-settings' => [
        User::class => [
            'filterableAttributes'=> ['id', 'name', 'email'],
            'sortableAttributes' => ['created_at'],
            // Other settings fields...
        ],
        Flight::class => [
            'filterableAttributes'=> ['id', 'destination'],
            'sortableAttributes' => ['updated_at'],
        ],
    ],
],
]
,
```

If the model underlying a given index is soft deletable and is included in the `index-settings` array, Scout will automatically include support for filtering on soft deleted models on that index. If you have no other filterable or sortable attributes to define for a soft deletable model index, you may simply add an empty entry to the `index-settings` array for that model:

```

'index-settings' => [
    Flight::class => []
],
]
```

After configuring your application's index settings, you must invoke the `scout:sync-index-settings` Artisan command. This command will inform Meilisearch of your currently configured index settings. For convenience, you may wish to make this command part of your deployment process:

```
php artisan scout:sync-index-settings
```

Configuring the Model ID

By default, Scout will use the primary key of the model as the model's unique ID / key that is stored in the search index. If you need to customize this behavior, you may override the `getScoutKey` and the `getScoutKeyName` methods on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * Get the value used to index the model.
     */
    public function getScoutKey(): mixed
    {
        return $this->email;
    }

    /**
     * Get the key name used to index the model.
     */
    public function getScoutKeyName(): mixed
    {
        return 'email';
    }
}
```

Configuring Search Engines per Model

When searching, Scout will typically use the default search engine specified in your application's `scout` configuration file. However, the search engine for a particular model can be changed by overriding the `searchableUsing` method on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Engines\Engine;
use Laravel\Scout\EngineManager;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * Get the engine used to index the model.
     */
    public function searchableUsing(): Engine
    {
        return app(EngineManager::class)->engine('meilisearch');
    }
}
```

Identifying Users

Scout also allows you to auto identify users when using [Algolia](#). Associating the authenticated user with search operations may be helpful when viewing your search analytics within Algolia's dashboard. You can enable user identification by defining a `SCOUT_IDENTIFY` environment variable as `true` in your application's `.env` file:

```
SCOUT_IDENTIFY=true
```

Enabling this feature will also pass the request's IP address and your authenticated user's primary identifier to Algolia so this data is associated with any search request that is made by the user.

Database / Collection Engines

Database Engine



The database engine currently supports MySQL and PostgreSQL.

If your application interacts with small to medium sized databases or has a light workload, you may find it more convenient to get started with Scout's "database" engine. The database engine will use "where like" clauses and full text indexes when filtering results from your existing database to determine the applicable search results for your query.

To use the database engine, you may simply set the value of the `SCOUT_DRIVER` environment variable to `database`, or specify the `database` driver directly in your application's `scout` configuration file:

```
SCOUT_DRIVER=database
```

Once you have specified the database engine as your preferred driver, you must [configure your searchable data](#). Then, you may start [executing search queries](#) against your models. Search engine indexing, such as the indexing needed to seed Algolia, Meilisearch or Typesense indexes, is unnecessary when using the database engine.

Customizing Database Searching Strategies

By default, the database engine will execute a "where like" query against every model attribute that you have [configured as searchable](#). However, in some situations, this may result in poor performance. Therefore, the database engine's search strategy can be configured so that some specified columns utilize full text search queries or only use "where like" constraints to search the prefixes of strings (`(example%)`) instead of searching within the entire string (`(%example%)`).

To define this behavior, you may assign PHP attributes to your model's `toSearchableArray` method. Any columns that are not assigned additional search strategy behavior will continue to use the default "where like" strategy:

```
use Laravel\Scout\Attributes\SearchUsingFullText;
use Laravel\Scout\Attributes\SearchUsingPrefix;

/**
 * Get the indexable data array for the model.
 *
 * @return array<string, mixed>
 */
#[SearchUsingPrefix(['id', 'email'])]
#[SearchUsingFullText(['bio'])]
public function toSearchableArray(): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'bio' => $this->bio,
    ];
}
```



Before specifying that a column should use full text query constraints, ensure that the column has been assigned a [full text index](#).

Collection Engine

While you are free to use the Algolia, Meilisearch, or Typesense search engines during local development, you may find it more convenient to get started with the "collection" engine. The collection engine will use "where" clauses and collection filtering on results from your existing database to determine the applicable search results for your query. When using this engine, it is not necessary to "index" your searchable models, as they will simply be retrieved from your local database.

To use the collection engine, you may simply set the value of the `SCOUT_DRIVER` environment variable to `collection`, or specify the `collection` driver directly in your application's `scout` configuration file:

```
SCOUT_DRIVER=collection
```

Once you have specified the collection driver as your preferred driver, you may start [executing search queries](#) against your models. Search engine indexing, such as the indexing needed to seed Algolia, Meilisearch, or Typesense indexes, is unnecessary when using the collection engine.

Differences From Database Engine

On first glance, the "database" and "collections" engines are fairly similar. They both interact directly with your database to retrieve search results. However, the collection engine does not utilize full text indexes or `LIKE` clauses to find

matching records. Instead, it pulls all possible records and uses Laravel's `Str::is` helper to determine if the search string exists within the model attribute values.

The collection engine is the most portable search engine as it works across all relational databases supported by Laravel (including SQLite and SQL Server); however, it is less efficient than Scout's database engine.

Indexing

Batch Import

If you are installing Scout into an existing project, you may already have database records you need to import into your indexes. Scout provides a `scout:import` Artisan command that you may use to import all of your existing records into your search indexes:

```
php artisan scout:import "App\Models\Post"
```

The `flush` command may be used to remove all of a model's records from your search indexes:

```
php artisan scout:flush "App\Models\Post"
```

Modifying the Import Query

If you would like to modify the query that is used to retrieve all of your models for batch importing, you may define a `makeAllSearchableUsing` method on your model. This is a great place to add any eager relationship loading that may be necessary before importing your models:

```
use Illuminate\Database\Eloquent\Builder;

/**
 * Modify the query used to retrieve models when making all of the models searchable.
 */
protected function makeAllSearchableUsing(Builder $query): Builder
{
    return $query->with('author');
}
```



The `makeAllSearchableUsing` method may not be applicable when using a queue to batch import models. Relationships are not restored when model collections are processed by jobs.

Adding Records

Once you have added the `Laravel\Scout\Searchable` trait to a model, all you need to do is `save` or `create` a model instance and it will automatically be added to your search index. If you have configured Scout to `use queues` this operation will be performed in the background by your queue worker:

```
use App\Models\Order;

$order = new Order;

// ...

$order->save();
```

Adding Records via Query

If you would like to add a collection of models to your search index via an Eloquent query, you may chain the `searchable` method onto the Eloquent query. The `searchable` method will chunk the results of the query and add the records to your search index. Again, if you have configured Scout to use queues, all of the chunks will be imported in the background by your queue workers:

```
use App\Models\Order;

Order::where('price', '>', 100)->searchable();
```

You may also call the `searchable` method on an Eloquent relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to add the model instances to their corresponding index:

```
$orders->searchable();
```



The `searchable` method can be considered an "upsert" operation. In other words, if the model record is already in your index, it will be updated. If it does not exist in the search index, it will be added to the index.

Updating Records

To update a searchable model, you only need to update the model instance's properties and `save` the model to your database. Scout will automatically persist the changes to your search index:

```
use App\Models\Order;

$order = Order::find(1);

// Update the order...

$order->save();
```

You may also invoke the `searchable` method on an Eloquent query instance to update a collection of models. If the models do not exist in your search index, they will be created:

```
Order::where('price', '>', 100)->searchable();
```

If you would like to update the search index records for all of the models in a relationship, you may invoke the `searchable` on the relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to update the model instances in their corresponding index:

```
$orders->searchable();
```

Modifying Records Before Importing

Sometimes you may need to prepare the collection of models before they are made searchable. For instance, you may want to eager load a relationship so that the relationship data can be efficiently added to your search index. To accomplish this, define a `makeSearchableUsing` method on the corresponding model:

```
use Illuminate\Database\Eloquent\Collection;

/**
 * Modify the collection of models being made searchable.
 */
public function makeSearchableUsing(Collection $models): Collection
{
    return $models->load('author');
}
```

Removing Records

To remove a record from your index you may simply `delete` the model from the database. This may be done even if you are using soft deleted models:

```
use App\Models\Order;

$order = Order::find(1);

$order->delete();
```

If you do not want to retrieve the model before deleting the record, you may use the `unsearchable` method on an Eloquent query instance:

```
Order::where('price', '>', 100)->unsearchable();
```

If you would like to remove the search index records for all of the models in a relationship, you may invoke the `unsearchable` on the relationship instance:

```
$user->orders()->unsearchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `unsearchable` method on the collection instance to remove the model instances from their corresponding index:

```
$orders->unsearchable();
```

Pausing Indexing

Sometimes you may need to perform a batch of Eloquent operations on a model without syncing the model data to your search index. You may do this using the `withoutSyncingToSearch` method. This method accepts a single closure which will be immediately executed. Any model operations that occur within the closure will not be synced to the model's index:

```
use App\Models\Order;

Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

Conditionally Searchable Model Instances

Sometimes you may need to only make a model searchable under certain conditions. For example, imagine you have `App\Models\Post` model that may be in one of two states: "draft" and "published". You may only want to allow "published" posts to be searchable. To accomplish this, you may define a `shouldBeSearchable` method on your model:

```
/** 
 * Determine if the model should be searchable.
 */
public function shouldBeSearchable(): bool
{
    return $this->isPublished();
}
```

The `shouldBeSearchable` method is only applied when manipulating models through the `save` and `create` methods, queries, or relationships. Directly making models or collections searchable using the `searchable` method will override the result of the `shouldBeSearchable` method.



The `shouldBeSearchable` method is not applicable when using Scout's "database" engine, as all searchable data is always stored in the database. To achieve similar behavior when using the database engine, you should use [where clauses](#) instead.

Searching

You may begin searching a model using the `search` method. The search method accepts a single string that will be used to search your models. You should then chain the `get` method onto the search query to retrieve the Eloquent models that match the given search query:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->get();
```

Since Scout searches return a collection of Eloquent models, you may even return the results directly from a route or controller and they will automatically be converted to JSON:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return Order::search($request->search)->get();
});
```

If you would like to get the raw search results before they are converted to Eloquent models, you may use the `raw` method:

```
$orders = Order::search('Star Trek')->raw();
```

Custom Indexes

Search queries will typically be performed on the index specified by the model's `searchableAs` method. However, you may use the `within` method to specify a custom index that should be searched instead:

```
$orders = Order::search('Star Trek')
->within('tv_shows_popularity_desc')
->get();
```

Where Clauses

Scout allows you to add simple "where" clauses to your search queries. Currently, these clauses only support basic numeric equality checks and are primarily useful for scoping search queries by an owner ID:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->where('user_id', 1)->get();
```

In addition, the `whereIn` method may be used to verify that a given column's value is contained within the given array:

```
$orders = Order::search('Star Trek')->whereIn(
    'status', ['open', 'paid']
)->get();
```

The `whereNotIn` method verifies that the given column's value is not contained in the given array:

```
$orders = Order::search('Star Trek')->whereNotIn(
    'status', ['closed']
)->get();
```

Since a search index is not a relational database, more advanced "where" clauses are not currently supported.



If your application is using Meilisearch, you must configure your application's [filterable attributes](#) before utilizing Scout's "where" clauses.

Pagination

In addition to retrieving a collection of models, you may paginate your search results using the `paginate` method. This method will return an `Illuminate\Pagination\LengthAwarePaginator` instance just as if you had [paginated a traditional Eloquent query](#):

```
use App\Models\Order;

$orders = Order::search('Star Trek')->paginate();
```

You may specify how many models to retrieve per page by passing the amount as the first argument to the `paginate` method:

```
$orders = Order::search('Star Trek')->paginate(15);
```

Once you have retrieved the results, you may display the results and render the page links using [Blade](#) just as if you had paginated a traditional Eloquent query:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

Of course, if you would like to retrieve the pagination results as JSON, you may return the paginator instance directly from a route or controller:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    return Order::search($request->input('query'))->paginate(15);
});
```



Since search engines are not aware of your Eloquent model's global scope definitions, you should not utilize global scopes in applications that utilize Scout pagination. Or, you should recreate the global scope's constraints when searching via Scout.

Soft Deleting

If your indexed models are [soft deleting](#) and you need to search your soft deleted models, set the `soft_delete` option of the `config/scout.php` configuration file to `true`:

```
'soft_delete' => true,
```

When this configuration option is `true`, Scout will not remove soft deleted models from the search index. Instead, it will set a hidden `_soft_deleted` attribute on the indexed record. Then, you may use the `withTrashed` or `onlyTrashed` methods to retrieve the soft deleted records when searching:

```
use App\Models\Order;

// Include trashed records when retrieving results...
$orders = Order::search('Star Trek')->withTrashed()->get();

// Only include trashed records when retrieving results...
$orders = Order::search('Star Trek')->onlyTrashed()->get();
```



When a soft deleted model is permanently deleted using `forceDelete`, Scout will remove it from the search index automatically.

Customizing Engine Searches

If you need to perform advanced customization of the search behavior of an engine you may pass a closure as the second argument to the `search` method. For example, you could use this callback to add geo-location data to your search options before the search query is passed to Algolia:

```
use Algolia\AlgoliaSearch\SearchIndex;
use App\Models\Order;

Order::search(
    'Star Trek',
    function (SearchIndex $algolia, string $query, array $options) {
        $options['body']['query']['bool']['filter']['geo_distance'] = [
            'distance' => '1000km',
            'location' => ['lat' => 36, 'lon' => 111],
        ];

        return $algolia->search($query, $options);
    }
)->get();
```

Customizing the Eloquent Results Query

After Scout retrieves a list of matching Eloquent models from your application's search engine, Eloquent is used to retrieve all of the matching models by their primary keys. You may customize this query by invoking the `query` method. The `query` method accepts a closure that will receive the Eloquent query builder instance as an argument:

```
use App\Models\Order;
use Illuminate\Database\Eloquent\Builder;

$orders = Order::search('Star Trek')
    ->query(fn (Builder $query) => $query->with('invoices'))
    ->get();
```

Since this callback is invoked after the relevant models have already been retrieved from your application's search engine, the `query` method should not be used for "filtering" results. Instead, you should use [Scout where clauses](#).

Custom Engines

Writing the Engine

If one of the built-in Scout search engines doesn't fit your needs, you may write your own custom engine and register it with Scout. Your engine should extend the `Laravel\Scout\Engines\Engine` abstract class. This abstract class contains eight methods your custom engine must implement:

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

You may find it helpful to review the implementations of these methods on the `Laravel\Scout\Engines\AlgoliaEngine` class. This class will provide you with a good starting point for learning how to implement each of these methods in your own engine.

Registering the Engine

Once you have written your custom engine, you may register it with Scout using the `extend` method of the Scout engine manager. Scout's engine manager may be resolved from the Laravel service container. You should call the `extend` method from the `boot` method of your `App\Providers\AppServiceProvider` class or any other service provider used by your application:

```
use App\ScoutExtensions\MySQLSearchEngine;
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySQLSearchEngine();
    });
}
```

Once your engine has been registered, you may specify it as your default Scout `driver` in your application's `config/scout.php` configuration file:

```
'driver' => 'mysql',
```

Laravel Socialite

- [Introduction](#)
- [Installation](#)
- [Upgrading Socialite](#)
- [Configuration](#)
- [Authentication](#)
 - [Routing](#)
 - [Authentication and Storage](#)
 - [Access Scopes](#)
 - [Slack Bot Scopes](#)
 - [Optional Parameters](#)
- [Retrieving User Details](#)

Introduction

In addition to typical, form based authentication, Laravel also provides a simple, convenient way to authenticate with OAuth providers using [Laravel Socialite](#). Socialite currently supports authentication via Facebook, Twitter, LinkedIn, Google, GitHub, GitLab, Bitbucket, and Slack.



Adapters for other platforms are available via the community driven [Socialite Providers](#) website.

Installation

To get started with Socialite, use the Composer package manager to add the package to your project's dependencies:

```
composer require laravel/socialite
```

Upgrading Socialite

When upgrading to a new major version of Socialite, it's important that you carefully review [the upgrade guide](#).

Configuration

Before using Socialite, you will need to add credentials for the OAuth providers your application utilizes. Typically, these credentials may be retrieved by creating a "developer application" within the dashboard of the service you will be authenticating with.

These credentials should be placed in your application's `config/services.php` configuration file, and should use the key `facebook`, `twitter` (OAuth 1.0), `twitter-oauth-2` (OAuth 2.0), `linkedin-openid`, `google`, `github`, `gitlab`, `bitbucket`, or `slack`, depending on the providers your application requires:

```
'github' => [
    'client_id' => env('GITHUB_CLIENT_ID'),
    'client_secret' => env('GITHUB_CLIENT_SECRET'),
    'redirect' => 'http://example.com/callback-url',
],
```



If the `redirect` option contains a relative path, it will automatically be resolved to a fully qualified URL.

Authentication

Routing

To authenticate users using an OAuth provider, you will need two routes: one for redirecting the user to the OAuth provider, and another for receiving the callback from the provider after authentication. The example routes below demonstrate the implementation of both routes:

```
use Laravel\\Socialite\\Facades\\Socialite;

Route::get('/auth/redirect', function () {
    return Socialite::driver('github')->redirect();
});

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // $user->token
});
```

The `redirect` method provided by the `Socialite` facade takes care of redirecting the user to the OAuth provider, while the `user` method will examine the incoming request and retrieve the user's information from the provider after they have approved the authentication request.

Authentication and Storage

Once the user has been retrieved from the OAuth provider, you may determine if the user exists in your application's database and [authenticate the user](#). If the user does not exist in your application's database, you will typically create a new record in your database to represent the user:

```

use App\Models\User;
use Illuminate\Support\Facades\Auth;
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/callback', function () {
    $githubUser = Socialite::driver('github')->user();

    $user = User::updateOrCreate([
        'github_id' => $githubUser->id,
    ], [
        'name' => $githubUser->name,
        'email' => $githubUser->email,
        'github_token' => $githubUser->token,
        'github_refresh_token' => $githubUser->refreshToken,
    ]);
});

Auth::login($user);

return redirect('/dashboard');
});

```



For more information regarding what user information is available from specific OAuth providers, please consult the documentation on [retrieving user details](#).

Access Scopes

Before redirecting the user, you may use the `scopes` method to specify the "scopes" that should be included in the authentication request. This method will merge all previously specified scopes with the scopes that you specify:

```

use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();

```

You can overwrite all existing scopes on the authentication request using the `setScopes` method:

```

return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();

```

Slack Bot Scopes

Slack's API provides [different types of access tokens](#), each with their own set of [permission scopes](#). Socialite is compatible with both of the following Slack access tokens types:

- Bot (prefixed with `xoxb-`)
- User (prefixed with `xoxp-`)

By default, the `slack` driver will generate a `user` token and invoking the driver's `user` method will return the user's details.

Bot tokens are primarily useful if your application will be sending notifications to external Slack workspaces that are owned by your application's users. To generate a bot token, invoke the `asBotUser` method before redirecting the user to Slack for authentication:

```
return Socialite::driver('slack')
    ->asBotUser()
    ->setScopes(['chat:write', 'chat:write.public', 'chat:write.customize'])
    ->redirect();
```

In addition, you must invoke the `asBotUser` method before invoking the `user` method after Slack redirects the user back to your application after authentication:

```
$user = Socialite::driver('slack')->asBotUser()->user();
```

When generating a bot token, the `user` method will still return a `Laravel\Socialite\Two\User` instance; however, only the `token` property will be hydrated. This token may be stored in order to [send notifications to the authenticated user's Slack workspaces](#).

Optional Parameters

A number of OAuth providers support other optional parameters on the redirect request. To include any optional parameters in the request, call the `with` method with an associative array:

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```



When using the `with` method, be careful not to pass any reserved keywords such as `state` or `response_type`.

Retrieving User Details

After the user is redirected back to your application's authentication callback route, you may retrieve the user's details using Socialite's `user` method. The user object returned by the `user` method provides a variety of properties and methods you may use to store information about the user in your own database.

Differing properties and methods may be available on this object depending on whether the OAuth provider you are authenticating with supports OAuth 1.0 or OAuth 2.0:

```
use Laravel\\Socialite\\Facades\\Socialite;

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // OAuth 2.0 providers...
    $token = $user->token;
    $refreshToken = $user->refreshToken;
    $expiresIn = $user->expiresIn;

    // OAuth 1.0 providers...
    $token = $user->token;
    $tokenSecret = $user->tokenSecret;

    // All providers...
    $user->getId();
    $user->getNickname();
    $user->getName();
    $user->getEmail();
    $user->getAvatar();
});

});
```

Retrieving User Details From a Token (OAuth2)

If you already have a valid access token for a user, you can retrieve their user details using Socialite's `userFromToken` method:

```
use Laravel\\Socialite\\Facades\\Socialite;

$user = Socialite::driver('github')->userFromToken($token);
```

Retrieving User Details From a Token and Secret (OAuth1)

If you already have a valid token and secret for a user, you can retrieve their user details using Socialite's `userFromTokenAndSecret` method:

```
use Laravel\\Socialite\\Facades\\Socialite;

$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);
```

Stateless Authentication

The `stateless` method may be used to disable session state verification. This is useful when adding social authentication to a stateless API that does not utilize cookie based sessions:

```
use Laravel\\Socialite\\Facades\\Socialite;

return Socialite::driver('google')->stateless()->user();
```



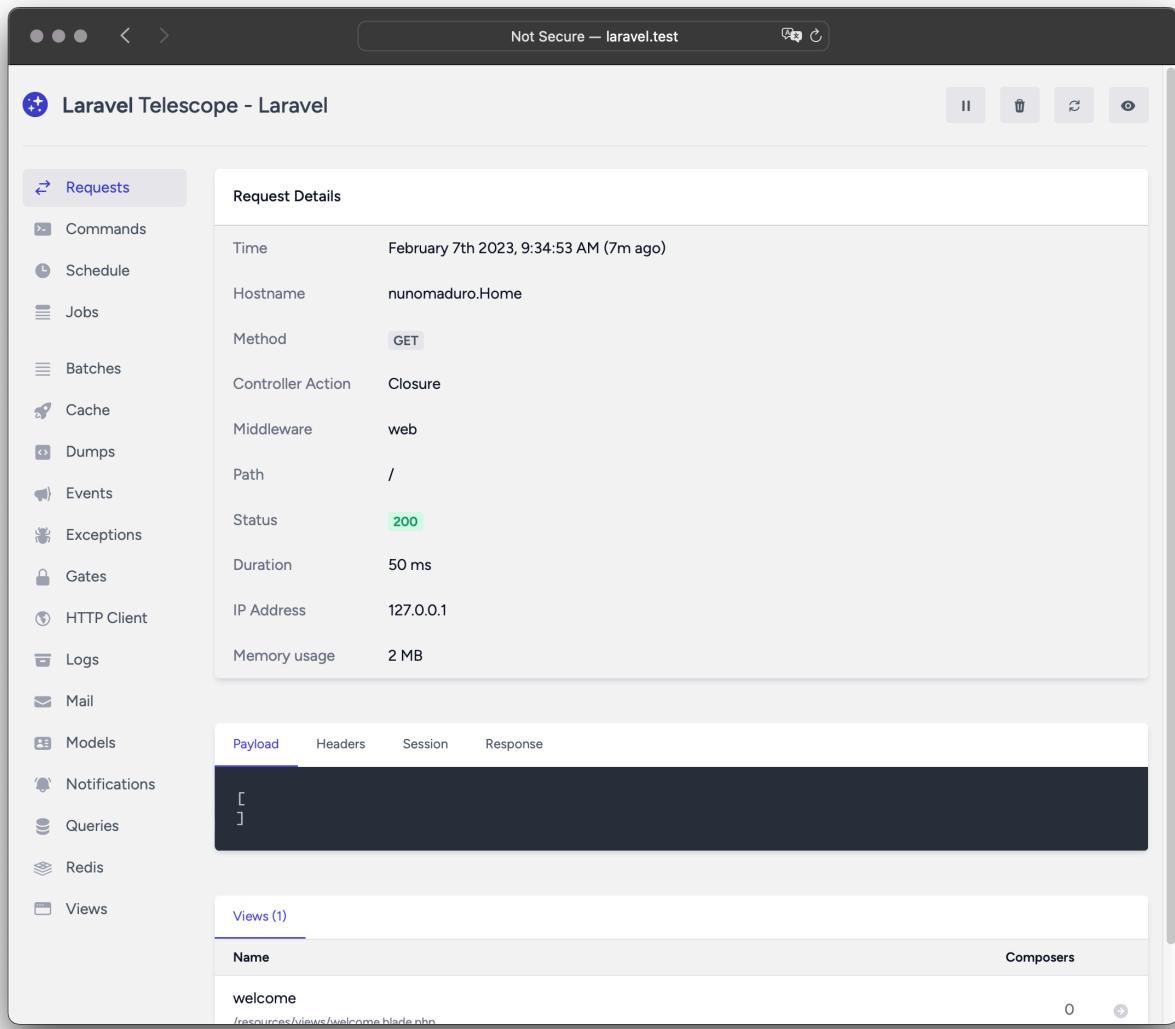
Stateless authentication is not available for the Twitter OAuth 1.0 driver.

Laravel Telescope

- [Introduction](#)
- [Installation](#)
 - [Local Only Installation](#)
 - [Configuration](#)
 - [Data Pruning](#)
 - [Dashboard Authorization](#)
- [Upgrading Telescope](#)
- [Filtering](#)
 - [Entries](#)
 - [Batches](#)
- [Tagging](#)
- [Available Watchers](#)
 - [Batch Watcher](#)
 - [Cache Watcher](#)
 - [Command Watcher](#)
 - [Dump Watcher](#)
 - [Event Watcher](#)
 - [Exception Watcher](#)
 - [Gate Watcher](#)
 - [HTTP Client Watcher](#)
 - [Job Watcher](#)
 - [Log Watcher](#)
 - [Mail Watcher](#)
 - [Model Watcher](#)
 - [Notification Watcher](#)
 - [Query Watcher](#)
 - [Redis Watcher](#)
 - [Request Watcher](#)
 - [Schedule Watcher](#)
 - [View Watcher](#)
- [Displaying User Avatars](#)

Introduction

[Laravel Telescope](#) makes a wonderful companion to your local Laravel development environment. Telescope provides insight into the requests coming into your application, exceptions, log entries, database queries, queued jobs, mail, notifications, cache operations, scheduled tasks, variable dumps, and more.



Installation

You may use the Composer package manager to install Telescope into your Laravel project:

```
composer require laravel/telescope
```

After installing Telescope, publish its assets using the `telescope:install` Artisan command. After installing Telescope, you should also run the `migrate` command in order to create the tables needed to store Telescope's data:

```
php artisan telescope:install
php artisan migrate
```

Finally, you may access the Telescope dashboard via the `/telescope` route.

Migration Customization

If you are not going to use Telescope's default migrations, you should call the `Telescope::ignoreMigrations` method in the `register` method of your application's `App\Providers\AppServiceProvider` class. You may export the default migrations using the following command: `php artisan vendor:publish --tag=telescope-migrations`

Local Only Installation

If you plan to only use Telescope to assist your local development, you may install Telescope using the `--dev` flag:

```
composer require laravel/telescope --dev

php artisan telescope:install

php artisan migrate
```

After running `telescope:install`, you should remove the `TelescopeServiceProvider` service provider registration from your application's `config/app.php` configuration file. Instead, manually register Telescope's service providers in the `register` method of your `App\Providers\AppServiceProvider` class. We will ensure the current environment is `local` before registering the providers:

```
/**
 * Register any application services.
 */
public function register(): void
{
    if ($this->app->environment('local')) {
        $this->app->register(\Laravel\Telescope\TelescopeServiceProvider::class);
        $this->app->register(TelescopeServiceProvider::class);
    }
}
```

Finally, you should also prevent the Telescope package from being auto-discovered by adding the following to your `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "laravel/telescope"
        ]
    }
},
```

Configuration

After publishing Telescope's assets, its primary configuration file will be located at `config/telescope.php`. This configuration file allows you to configure your watcher options. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.

If desired, you may disable Telescope's data collection entirely using the `enabled` configuration option:

```
'enabled' => env('TELESCOPE_ENABLED', true),
```

Data Pruning

Without pruning, the `telescope_entries` table can accumulate records very quickly. To mitigate this, you should schedule the `telescope:prune` Artisan command to run daily:

```
$schedule->command('telescope:prune')->daily();
```

By default, all entries older than 24 hours will be pruned. You may use the `hours` option when calling the command to determine how long to retain Telescope data. For example, the following command will delete all records created over 48 hours ago:

```
$schedule->command('telescope:prune --hours=48')->daily();
```

Dashboard Authorization

The Telescope dashboard may be accessed via the `/telescope` route. By default, you will only be able to access this dashboard in the `local` environment. Within your `app/Providers/TelescopeServiceProvider.php` file, there is an [authorization gate](#) definition. This authorization gate controls access to Telescope in `non-local` environments. You are free to modify this gate as needed to restrict access to your Telescope installation:

```
use App\Models\User;

/**
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope in non-local environments.
 */
protected function gate(): void
{
    Gate::define('viewTelescope', function (User $user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```



You should ensure you change your `APP_ENV` environment variable to `production` in your production environment. Otherwise, your Telescope installation will be publicly available.

Upgrading Telescope

When upgrading to a new major version of Telescope, it's important that you carefully review [the upgrade guide](#).

In addition, when upgrading to any new Telescope version, you should re-publish Telescope's assets:

```
php artisan telescope:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `vendor:publish --tag=laravel-assets` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan vendor:publish --tag=laravel-assets --ansi --force"
        ]
    }
}
```

Filtering

Entries

You may filter the data that is recorded by Telescope via the `filter` closure that is defined in your `App\Providers\TelescopeServiceProvider` class. By default, this closure records all data in the `local` environment and exceptions, failed jobs, scheduled tasks, and data with monitored tags in all other environments:

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::filter(function (IncomingEntry $entry) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entry->isReportableException() ||
            $entry->isFailedJob() ||
            $entry->isScheduledTask() ||
            $entry->isSlowQuery() ||
            $entry->hasMonitoredTag();
    });
}
```

Batches

While the `filter` closure filters data for individual entries, you may use the `filterBatch` method to register a closure that filters all data for a given request or console command. If the closure returns `true`, all of the entries are recorded by Telescope:

```

use Illuminate\Support\Collection;
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::filterBatch(function (Collection $entries) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entries->contains(function (IncomingEntry $entry) {
            return $entry->isReportableException() ||
                $entry->isFailedJob() ||
                $entry->isScheduledTask() ||
                $entry->isSlowQuery() ||
                $entry->hasMonitoredTag();
        });
    });
}

```

Tagging

Telescope allows you to search entries by "tag". Often, tags are Eloquent model class names or authenticated user IDs which Telescope automatically adds to entries. Occasionally, you may want to attach your own custom tags to entries. To accomplish this, you may use the `Telescope::tag` method. The `tag` method accepts a closure which should return an array of tags. The tags returned by the closure will be merged with any tags Telescope would automatically attach to the entry. Typically, you should call the `tag` method within the `register` method of your `App\Providers\TelescopeServiceProvider` class:

```

use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::tag(function (IncomingEntry $entry) {
        return $entry->type === 'request'
            ? ['status' . $entry->content['response_status']]
            : [];
    });
}

```

Available Watchers

Telescope "watchers" gather application data when a request or console command is executed. You may customize the list of watchers that you would like to enable within your `config/telescope.php` configuration file:

```
'watchers' => [
    Watchers\CacheWatcher::class => true,
    Watchers\CommandWatcher::class => true,
    ...
],
```

Some watchers also allow you to provide additional customization options:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 100,
    ],
    ...
],
```

Batch Watcher

The batch watcher records information about queued [batches](#), including the job and connection information.

Cache Watcher

The cache watcher records data when a cache key is hit, missed, updated and forgotten.

Command Watcher

The command watcher records the arguments, options, exit code, and output whenever an Artisan command is executed. If you would like to exclude certain commands from being recorded by the watcher, you may specify the command in the `ignore` option within your `config/telescope.php` file:

```
'watchers' => [
    Watchers\CommandWatcher::class => [
        'enabled' => env('TELESCOPE_COMMAND_WATCHER', true),
        'ignore' => ['key:generate'],
    ],
    ...
],
```

Dump Watcher

The dump watcher records and displays your variable dumps in Telescope. When using Laravel, variables may be dumped using the global `dump` function. The dump watcher tab must be open in a browser for the dump to be recorded, otherwise, the dumps will be ignored by the watcher.

Event Watcher

The event watcher records the payload, listeners, and broadcast data for any [events](#) dispatched by your application. The Laravel framework's internal events are ignored by the Event watcher.

Exception Watcher

The exception watcher records the data and stack trace for any reportable exceptions that are thrown by your application.

Gate Watcher

The gate watcher records the data and result of [gate and policy](#) checks by your application. If you would like to exclude certain abilities from being recorded by the watcher, you may specify those in the `ignore_abilities` option in your `config/telescope.php` file:

```
'watchers' => [
    Watchers\GateWatcher::class => [
        'enabled' => env('TELESCOPE_GATE_WATCHER', true),
        'ignore_abilities' => ['viewNova'],
    ],
    ...
],
```

HTTP Client Watcher

The HTTP client watcher records outgoing [HTTP client requests](#) made by your application.

Job Watcher

The job watcher records the data and status of any [jobs](#) dispatched by your application.

Log Watcher

The log watcher records the [log data](#) for any logs written by your application.

By default, Telescope will only record logs at the `error` level and above. However, you can modify the `level` option in your application's `config/telescope.php` configuration file to modify this behavior:

```
'watchers' => [
    Watchers\LogWatcher::class => [
        'enabled' => env('TELESCOPE_LOG_WATCHER', true),
        'level' => 'debug',
    ],
    ...
],
```

Mail Watcher

The mail watcher allows you to view an in-browser preview of [emails](#) sent by your application along with their associated data. You may also download the email as an `.eml` file.

Model Watcher

The model watcher records model changes whenever an Eloquent [model event](#) is dispatched. You may specify which model events should be recorded via the watcher's `events` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
    ],
    ...
],
```

If you would like to record the number of models hydrated during a given request, enable the `hydrations` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
        'hydrations' => true,
    ],
    ...
],
```

Notification Watcher

The notification watcher records all [notifications](#) sent by your application. If the notification triggers an email and you have the mail watcher enabled, the email will also be available for preview on the mail watcher screen.

Query Watcher

The query watcher records the raw SQL, bindings, and execution time for all queries that are executed by your application. The watcher also tags any queries slower than 100 milliseconds as `slow`. You may customize the slow query threshold using the watcher's `slow` option:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 50,
    ],
    ...
],
```

Redis Watcher

The Redis watcher records all [Redis](#) commands executed by your application. If you are using Redis for caching, cache commands will also be recorded by the Redis watcher.

Request Watcher

The request watcher records the request, headers, session, and response data associated with any requests handled by the application. You may limit your recorded response data via the `size_limit` (in kilobytes) option:

```
'watchers' => [
    Watchers\RequestWatcher::class => [
        'enabled' => env('TELESCOPE_REQUEST_WATCHER', true),
        'size_limit' => env('TELESCOPE_RESPONSE_SIZE_LIMIT', 64),
    ],
    ...
],
```

Schedule Watcher

The schedule watcher records the command and output of any [scheduled tasks](#) run by your application.

View Watcher

The view watcher records the `view` name, path, data, and "composers" used when rendering views.

Displaying User Avatars

The Telescope dashboard displays the user avatar for the user that was authenticated when a given entry was saved. By default, Telescope will retrieve avatars using the Gravatar web service. However, you may customize the avatar URL by registering a callback in your `App\Providers\TelescopeServiceProvider` class. The callback will receive the user's ID and email address and should return the user's avatar image URL:

```
use App\Models\User;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 */
public function register(): void
{
    // ...

    Telescope::avatar(function (string $id, string $email) {
        return '/avatars/'.User::find($id)->avatar_path;
    });
}
```

Laravel Valet

- [Introduction](#)
- [Installation](#)
 - [Upgrading Valet](#)
- [Serving Sites](#)
 - [The "Park" Command](#)
 - [The "Link" Command](#)
 - [Securing Sites With TLS](#)
 - [Serving a Default Site](#)
 - [Per-Site PHP Versions](#)
- [Sharing Sites](#)
 - [Sharing Sites on Your Local Network](#)
- [Site Specific Environment Variables](#)
- [Proxying Services](#)
- [Custom Valet Drivers](#)
 - [Local Drivers](#)
- [Other Valet Commands](#)
- [Valet Directories and Files](#)
 - [Disk Access](#)

Introduction



Looking for an even easier way to develop Laravel applications on macOS? Check out [Laravel Herd](#). Herd includes everything you need to get started with Laravel development, including Valet, PHP, and Composer.

[Laravel Valet](#) is a development environment for macOS minimalists. Laravel Valet configures your Mac to always run [Nginx](#) in the background when your machine starts. Then, using [DnsMasq](#), Valet proxies all requests on the `*.test` domain to point to sites installed on your local machine.

In other words, Valet is a blazing fast Laravel development environment that uses roughly 7 MB of RAM. Valet isn't a complete replacement for [Sail](#) or [Homestead](#), but provides a great alternative if you want flexible basics, prefer extreme speed, or are working on a machine with a limited amount of RAM.

Out of the box, Valet support includes, but is not limited to:

- | | | |
|-------------------------------|------------------------------------|-----------------------------|
| • Laravel | • ExpressionEngine | • Sculpin |
| • Bedrock | • Jigsaw | • Slim |
| • CakePHP 3 | • Joomla | • Statamic |
| • ConcreteCMS | • Katana | • Static HTML |
| • Contao | • Kirby | • Symfony |
| • Craft | • Magento | • WordPress |
| • Drupal | • OctoberCMS | • Zend |

However, you may extend Valet with your own [custom drivers](#).

Installation



Valet requires macOS and [Homebrew](#). Before installation, you should make sure that no other programs such as Apache or Nginx are binding to your local machine's port 80.

To get started, you first need to ensure that Homebrew is up to date using the `update` command:

```
brew update
```

Next, you should use Homebrew to install PHP:

```
brew install php
```

After installing PHP, you are ready to install the [Composer package manager](#). In addition, you should make sure the `$HOME/.composer/vendor/bin` directory is in your system's "PATH". After Composer has been installed, you may install Laravel Valet as a global Composer package:

```
composer global require laravel/valet
```

Finally, you may execute Valet's `install` command. This will configure and install Valet and DnsMasq. In addition, the daemons Valet depends on will be configured to launch when your system starts:

```
valet install
```

Once Valet is installed, try pinging any `*.test` domain on your terminal using a command such as `ping foobar.test`. If Valet is installed correctly you should see this domain responding on `127.0.0.1`.

Valet will automatically start its required services each time your machine boots.

PHP Versions



Instead of modifying your global PHP version, you can instruct Valet to use per-site PHP versions via the `isolate` command.

Valet allows you to switch PHP versions using the `valet use php@version` command. Valet will install the specified PHP version via Homebrew if it is not already installed:

```
valet use php@8.1
```

```
valet use php
```

You may also create a `.valetrc` file in the root of your project. The `.valetrc` file should contain the PHP version the site should use:

```
php=php@8.1
```

Once this file has been created, you may simply execute the `valet use` command and the command will determine the site's preferred PHP version by reading the file.



Valet only serves one PHP version at a time, even if you have multiple PHP versions installed.

Database

If your application needs a database, check out [DBngin](#), which provides a free, all-in-one database management tool that includes MySQL, PostgreSQL, and Redis. After DBngin has been installed, you can connect to your database at `127.0.0.1` using the `root` username and an empty string for the password.

Resetting Your Installation

If you are having trouble getting your Valet installation to run properly, executing the `composer global require laravel/valet` command followed by `valet install` will reset your installation and can solve a variety of problems. In rare cases, it may be necessary to "hard reset" Valet by executing `valet uninstall --force` followed by `valet install`.

Upgrading Valet

You may update your Valet installation by executing the `composer global require laravel/valet` command in your terminal. After upgrading, it is good practice to run the `valet install` command so Valet can make additional upgrades to your configuration files if necessary.

Upgrading to Valet 4

If you're upgrading from Valet 3 to Valet 4, take the following steps to properly upgrade your Valet installation:

- If you've added `.valetphprc` files to customize your site's PHP version, rename each `.valetphprc` file to `.valetrc`. Then, prepend `php=` to the existing content of the `.valetrc` file.
- Update any custom drivers to match the namespace, extension, type-hints, and return type-hints of the new driver system. You may consult Valet's [SampleValetDriver](#) as an example.
- If you use PHP 7.1 - 7.4 to serve your sites, make sure you still use Homebrew to install a version of PHP that's 8.0 or higher, as Valet will use this version, even if it's not your primary linked version, to run some of its scripts.

Serving Sites

Once Valet is installed, you're ready to start serving your Laravel applications. Valet provides two commands to help you serve your applications: `park` and `link`.

The `park` Command

The `park` command registers a directory on your machine that contains your applications. Once the directory has been "parked" with Valet, all of the directories within that directory will be accessible in your web browser at `http://<directory-name>.test`:

```
cd ~/Sites
valet park
```

That's all there is to it. Now, any application you create within your "parked" directory will automatically be served using the `http://<directory-name>.test` convention. So, if your parked directory contains a directory named "laravel", the application within that directory will be accessible at `http://laravel.test`. In addition, Valet automatically allows you to access the site using wildcard subdomains (`http://foo.laravel.test`).

The `link` Command

The `link` command can also be used to serve your Laravel applications. This command is useful if you want to serve a single site in a directory and not the entire directory:

```
cd ~/Sites/laravel
valet link
```

Once an application has been linked to Valet using the `link` command, you may access the application using its directory name. So, the site that was linked in the example above may be accessed at `http://laravel.test`. In addition, Valet automatically allows you to access the site using wildcard sub-domains (`http://foo.laravel.test`).

If you would like to serve the application at a different hostname, you may pass the hostname to the `link` command. For example, you may run the following command to make an application available at `http://application.test`:

```
cd ~/Sites/laravel
valet link application
```

Of course, you may also serve applications on subdomains using the `link` command:

```
valet link api.application
```

You may execute the `links` command to display a list of all of your linked directories:

```
valet links
```

The `unlink` command may be used to destroy the symbolic link for a site:

```
cd ~/Sites/laravel
valet unlink
```

Securing Sites With TLS

By default, Valet serves sites over HTTP. However, if you would like to serve a site over encrypted TLS using HTTP/2, you may use the `secure` command. For example, if your site is being served by Valet on the `laravel.test` domain, you should run the following command to secure it:

```
valet secure laravel
```

To "unsecure" a site and revert back to serving its traffic over plain HTTP, use the `unsecure` command. Like the `secure` command, this command accepts the hostname that you wish to unsecure:

```
valet unsecure laravel
```

Serving a Default Site

Sometimes, you may wish to configure Valet to serve a "default" site instead of a `404` when visiting an unknown `test` domain. To accomplish this, you may add a `default` option to your `~/.config/valet/config.json` configuration file containing the path to the site that should serve as your default site:

```
"default": "/Users/Sally/Sites/example-site",
```

Per-Site PHP Versions

By default, Valet uses your global PHP installation to serve your sites. However, if you need to support multiple PHP versions across various sites, you may use the `isolate` command to specify which PHP version a particular site should use. The `isolate` command configures Valet to use the specified PHP version for the site located in your current working directory:

```
cd ~/Sites/example-site
valet isolate php@8.0
```

If your site name does not match the name of the directory that contains it, you may specify the site name using the `--site` option:

```
valet isolate php@8.0 --site="site-name"
```

For convenience, you may use the `valet php`, `composer`, and `which-php` commands to proxy calls to the appropriate PHP CLI or tool based on the site's configured PHP version:

```
valet php
valet composer
valet which-php
```

You may execute the `isolated` command to display a list of all of your isolated sites and their PHP versions:

```
valet isolated
```

To revert a site back to Valet's globally installed PHP version, you may invoke the `unisolate` command from the site's root directory:

```
valet unisolate
```

Sharing Sites

Valet includes a command to share your local sites with the world, providing an easy way to test your site on mobile devices or share it with team members and clients.

Out of the box, Valet supports sharing your sites via ngrok or Expose. Before sharing a site, you should update your Valet configuration using the `share-tool` command, specifying either `ngrok` or `expose`:

```
valet share-tool ngrok
```

If you choose a tool and don't have it installed via Homebrew (for ngrok) or Composer (for Expose), Valet will automatically prompt you to install it. Of course, both tools require you to authenticate your ngrok or Expose account before you can start sharing sites.

To share a site, navigate to the site's directory in your terminal and run Valet's `share` command. A publicly accessible URL will be placed into your clipboard and is ready to paste directly into your browser or to be shared with your team:

```
cd ~/Sites/laravel
```

```
valet share
```

To stop sharing your site, you may press `Control + C`.



If you're using a custom DNS server (like `1.1.1.1`), ngrok sharing may not work correctly. If this is the case on your machine, open your Mac's system settings, go to the Network settings, open the Advanced settings, then go the DNS tab and add `127.0.0.1` as your first DNS server.

Sharing Sites via Ngrok

Sharing your site using ngrok requires you to [create an ngrok account](#) and [set up an authentication token](#). Once you have an authentication token, you can update your Valet configuration with that token:

```
valet set-ngrok-token YOUR_TOKEN_HERE
```



You may pass additional ngrok parameters to the share command, such as `valet share --region=eu`. For more information, consult the [ngrok documentation](#).

Sharing Sites via Expose

Sharing your site using Expose requires you to [create an Expose account](#) and [authenticate with Expose via your authentication token](#).

You may consult the [Expose documentation](#) for information regarding the additional command-line parameters it supports.

Sharing Sites on Your Local Network

Valet restricts incoming traffic to the internal `127.0.0.1` interface by default so that your development machine isn't exposed to security risks from the Internet.

If you wish to allow other devices on your local network to access the Valet sites on your machine via your machine's IP address (eg: `192.168.1.10/application.test`), you will need to manually edit the appropriate Nginx configuration file for that site to remove the restriction on the `listen` directive. You should remove the `127.0.0.1:` prefix on the `listen` directive for ports 80 and 443.

If you have not run `valet secure` on the project, you can open up network access for all non-HTTPS sites by editing the `/usr/local/etc/nginx/valet/valet.conf` file. However, if you're serving the project site over HTTPS (you have run `valet secure` for the site) then you should edit the `~/.config/valet/Nginx/app-name.test` file.

Once you have updated your Nginx configuration, run the `valet restart` command to apply the configuration changes.

Site Specific Environment Variables

Some applications using other frameworks may depend on server environment variables but do not provide a way for those variables to be configured within your project. Valet allows you to configure site specific environment variables by adding a `.valet-env.php` file within the root of your project. This file should return an array of site / environment variable pairs which will be added to the global `$_SERVER` array for each site specified in the array:

```
<?php

return [
    // Set $_SERVER['key'] to "value" for the laravel.test site...
    'laravel' => [
        'key' => 'value',
    ],
    // Set $_SERVER['key'] to "value" for all sites...
    '*' => [
        'key' => 'value',
    ],
];
```

Proxying Services

Sometimes you may wish to proxy a Valet domain to another service on your local machine. For example, you may occasionally need to run Valet while also running a separate site in Docker; however, Valet and Docker can't both bind to port 80 at the same time.

To solve this, you may use the `proxy` command to generate a proxy. For example, you may proxy all traffic from `http://elasticsearch.test` to `http://127.0.0.1:9200`:

```
# Proxy over HTTP...
valet proxy elasticsearch http://127.0.0.1:9200

# Proxy over TLS + HTTP/2...
valet proxy elasticsearch http://127.0.0.1:9200 --secure
```

You may remove a proxy using the `unproxy` command:

```
valet unproxy elasticsearch
```

You may use the `proxies` command to list all site configurations that are proxied:

```
valet proxies
```

Custom Valet Drivers

You can write your own Valet "driver" to serve PHP applications running on a framework or CMS that is not natively supported by Valet. When you install Valet, a `~/.config/valet/Drivers` directory is created which contains a `SampleValetDriver.php` file. This file contains a sample driver implementation to demonstrate how to write a custom driver. Writing a driver only requires you to implement three methods: `serves`, `isStaticFile`, and `frontControllerPath`.

All three methods receive the `$sitePath`, `$siteName`, and `$uri` values as their arguments. The `$sitePath` is the fully qualified path to the site being served on your machine, such as `/Users/Lisa/Sites/my-project`. The `$siteName` is the "host" / "site name" portion of the domain (`my-project`). The `$uri` is the incoming request URI (`/foo/bar`).

Once you have completed your custom Valet driver, place it in the `~/.config/valet/Drivers` directory using the `FrameworkValetDriver.php` naming convention. For example, if you are writing a custom valet driver for WordPress, your filename should be `WordPressValetDriver.php`.

Let's take a look at a sample implementation of each method your custom Valet driver should implement.

The `serves` Method

The `serves` method should return `true` if your driver should handle the incoming request. Otherwise, the method should return `false`. So, within this method, you should attempt to determine if the given `$sitePath` contains a project of the type you are trying to serve.

For example, let's imagine we are writing a `WordPressValetDriver`. Our `serves` method might look something like this:

```
/**
 * Determine if the driver serves the request.
 */
public function serves(string $sitePath, string $siteName, string $uri): bool
{
    return is_dir($sitePath . '/wp-admin');
}
```

The `isStaticFile` Method

The `isStaticFile` should determine if the incoming request is for a file that is "static", such as an image or a stylesheet. If the file is static, the method should return the fully qualified path to the static file on disk. If the incoming request is not for a static file, the method should return `false`:

```
/**
 * Determine if the incoming request is for a static file.
 *
 * @return string|false
 */
public function isStaticFile(string $sitePath, string $siteName, string $uri)
{
    if (file_exists($staticFilePath = $sitePath . '/public/' . $uri)) {
        return $staticFilePath;
    }

    return false;
}
```



The `isStaticFile` method will only be called if the `serves` method returns `true` for the incoming request and the request URI is not `/`.

The `frontControllerPath` Method

The `frontControllerPath` method should return the fully qualified path to your application's "front controller", which is typically an "index.php" file or equivalent:

```
/**  
 * Get the fully resolved path to the application's front controller.  
 */  
public function frontControllerPath(string $sitePath, string $siteName, string $uri): string  
{  
    return $sitePath.'/public/index.php';  
}
```

Local Drivers

If you would like to define a custom Valet driver for a single application, create a `LocalValetDriver.php` file in the application's root directory. Your custom driver may extend the base `ValetDriver` class or extend an existing application specific driver such as the `LaravelValetDriver`:

```
use Valet\Drivers\LaravelValetDriver;  
  
class LocalValetDriver extends LaravelValetDriver  
{  
    /**  
     * Determine if the driver serves the request.  
     */  
    public function serves(string $sitePath, string $siteName, string $uri): bool  
    {  
        return true;  
    }  
  
    /**  
     * Get the fully resolved path to the application's front controller.  
     */  
    public function frontControllerPath(string $sitePath, string $siteName, string $uri): string  
    {  
        return $sitePath.'/public_html/index.php';  
    }  
}
```

Other Valet Commands

Command	Description
valet list	Display a list of all Valet commands.
valet diagnose	Output diagnostics to aid in debugging Valet.
valet directory-listing	Determine directory-listing behavior. Default is "off", which renders a 404 page for directories.
valet forget	Run this command from a "parked" directory to remove it from the parked directory list.
valet log	View a list of logs which are written by Valet's services.
valet paths	View all of your "parked" paths.
valet restart	Restart the Valet daemons.
valet start	Start the Valet daemons.
valet stop	Stop the Valet daemons.
valet trust	Add sudoers files for Brew and Valet to allow Valet commands to be run without prompting for your password.
valet uninstall	Uninstall Valet: shows instructions for manual uninstall. Pass the --force option to aggressively delete all of Valet's resources.

Valet Directories and Files

You may find the following directory and file information helpful while troubleshooting issues with your Valet environment:

~/.config/valet

Contains all of Valet's configuration. You may wish to maintain a backup of this directory.

~/.config/valet/dnsmasq.d/

This directory contains DNSMasq's configuration.

~/.config/valet/Drivers/

This directory contains Valet's drivers. Drivers determine how a particular framework / CMS is served.

~/.config/valet/Nginx/

This directory contains all of Valet's Nginx site configurations. These files are rebuilt when running the `install` and `secure` commands.

~/.config/valet/Sites/

This directory contains all of the symbolic links for your [linked projects](#).

~/.config/valet/config.json

This file is Valet's master configuration file.

~/.config/valet/valet.sock

This file is the PHP-FPM socket used by Valet's Nginx installation. This will only exist if PHP is running properly.

~/.config/valet/Log/fpm-php.www.log

This file is the user log for PHP errors.

`~/.config/valet/Log/nginx-error.log`

This file is the user log for Nginx errors.

`/usr/local/var/log/php-fpm.log`

This file is the system log for PHP-FPM errors.

`/usr/local/var/log/nginx`

This directory contains the Nginx access and error logs.

`/usr/local/etc/php/X.X/conf.d`

This directory contains the `*.ini` files for various PHP configuration settings.

`/usr/local/etc/php/X.X/php-fpm.d/valet-fpm.conf`

This file is the PHP-FPM pool configuration file.

`~/.composer/vendor/laravel/valet/cli/stubs/secure.valet.conf`

This file is the default Nginx configuration used for building SSL certificates for your sites.

Disk Access

Since macOS 10.14, [access to some files and directories is restricted by default](#). These restrictions include the Desktop, Documents, and Downloads directories. In addition, network volume and removable volume access is restricted. Therefore, Valet recommends your site folders are located outside of these protected locations.

However, if you wish to serve sites from within one of those locations, you will need to give Nginx "Full Disk Access". Otherwise, you may encounter server errors or other unpredictable behavior from Nginx, especially when serving static assets. Typically, macOS will automatically prompt you to grant Nginx full access to these locations. Or, you may do so manually via `System Preferences` > `Security & Privacy` > `Privacy` and selecting `Full Disk Access`. Next, enable any `nginx` entries in the main window pane.