Mastering Laravel 12.x - A Comprehensive Guide

This book is your comprehensive guide to mastering Laravel, the popular PHP framework for web artisans. It covers everything from basic setup to advanced concepts, providing in-depth knowledge for developers of all levels.

Table of Contents

- 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
- MongoDB

Eloquent ORM

- Getting Started
- Relationships
- Collections
- Mutators / Casts
- API Resources
- Serialization
- Factories

Testing

- Getting Started
- HTTP Tests
- Console Tests
- Browser Tests
- Database
- Mocking

Packages

- Cashier (Stripe)
- Cashier (Paddle)
- Dusk
- Envoy
- Fortify

- Folio
- Homestead
- Horizon
- Mix
- Octane
- Passport
- Pennant
- Pint
- Precognition
- Prompts
- Pulse
- Reverb
- Sail
- Sanctum
- Scout
- Socialite
- Telescope
- Valet

Release Notes

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 `^12.0`, 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, 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
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
12	8.2 - 8.4	February 24th, 2025	August 13th, 2026	February 24th, 2027
13	8.3 - 8.4	Q1 2026	Q3 2027	Q1 2028

(*) Supported PHP versions

End of life and *Security fixes only* periods are indicated in the table.

Laravel 12

Laravel 12 continues the improvements made in Laravel 11.x by updating upstream dependencies and introducing new starter kits for React, Vue, and Livewire, including the option of using WorkOS AuthKit for user authentication. The WorkOS variant of our starter kits offers social authentication, passkeys, and SSO support.

Minimal Breaking Changes

Much of our focus during this release cycle has been minimizing breaking changes. Instead, we have dedicated ourselves to shipping continuous quality-of-life improvements throughout the year that do not break existing applications.

Therefore, the Laravel 12 release is a relatively minor "maintenance release" in order to upgrade existing dependencies. Most Laravel applications may upgrade to Laravel 12 without changing any application code.

New Application Starter Kits

Laravel 12 introduces new starter kits for React, Vue, and Livewire. The React and Vue starter kits utilize Inertia 2, TypeScript, shadcn/ui, and Tailwind, while the Livewire starter kits utilize the Tailwind-based Flux UI component library and Laravel Volt.

All these starter kits utilize Laravel's built-in authentication system to offer login, registration, password reset, email verification, and more. Additionally, we are introducing a WorkOS AuthKit-powered variant of each starter kit, offering social authentication, passkeys, and SSO support. WorkOS provides free authentication for applications up to 1 million monthly active users.

With the introduction of these new application starter kits, Laravel Breeze and Laravel Jetstream will no longer receive additional updates.

To get started with our new starter kits, see the starter kit documentation.

Upgrade Guide

Upgrading To 12.0 From 11.x

Back to top

High Impact Changes

Medium Impact Changes

Low Impact Changes

Upgrading To 12.0 From 11.x

Estimated Upgrade Time: 5 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 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

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

- `laravel/framework` to `^12.0`
- `phpunit/phpunit` to `^11.0`
- `pestphp/pest` to `^3.0`

Carbon 3

Likelihood Of Impact: Low

Support for Carbon 2.x has been removed. All Laravel 12 applications now require Carbon 3.x.

Updating the Laravel Installer

If you're using the Laravel installer CLI tool to create new Laravel applications, update it to be compatible with Laravel 12.x and the new Laravel starter kits. If installed via `composer global require`, update with:

```
composer global update laravel/installer
```

If you installed PHP and Laravel via `php.new`, re-run the installer commands for your OS:

macOS

```
/bin/bash -c "$(curl -fsSL https://php.new/install/mac/8.4)"
```

Windows PowerShell

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::Se
```

Linux

```
/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.4)"
```

Or, if using Laravel Herd, update your Herd installation to the latest release.

Authentication

Updated `DatabaseTokenRepository` Constructor Signature

Likelihood Of Impact: Very Low

The constructor of `Illuminate\Auth\Passwords\DatabaseTokenRepository` now expects `$expires` in seconds, not minutes.

Concurrency

Concurrency Result Index Mapping

Likelihood Of Impact: Low

When using `Concurrency::run` with an associative array, results are now returned with their keys:

```
$result = Concurrency::run([
    'task-1' => fn () => 1 + 1,
    'task-2' => fn () => 2 + 2,
]);

// Results:
[
    'task-1' => 2,
    'task-2' => 4,
]
```

Container

Container Class Dependency Resolution

Likelihood Of Impact: Low

The container respects default property values when resolving classes. If previously relying on container to resolve classes without defaults, adjust accordingly:

```
class Example
{
    public function __construct(public ?Carbon $date = null) {}
}

$example = resolve(Example::class);

// <= 11.x
$example->date instanceof Carbon;

// >= 12.x
$example->date === null;
```

Database

Multi-Schema Database Inspecting

Likelihood Of Impact: Low

Methods like `Schema::getTables()`, `Schema::getViews()`, and `Schema::getTypes()` include all schemas by default. You can specify schema:

```
// All schemas
$tables = Schema::getTables();

// Specific schema
$tables = Schema::getTables(schema: 'main');

// Multiple schemas
$tables = Schema::getTables(schema: ['main', 'blog']);
```

`Schema::getTableListing()` now returns schema-qualified names by default. Use `schemaQualified` parameter to change:

```
$tables = Schema::getTableListing(); // ['main.migrations', 'main.users', 'blog.posts'
$tables = Schema::getTableListing(schema: 'main'); // ['main.migrations', 'main.users'
$tables = Schema::getTableListing(schema: 'main', schemaQualified: false); // ['migrat
```

Commands like `db:table` and `db:show` now show results across all schemas on MySQL, MariaDB, and SQLite.

Updated `Blueprint` Constructor Signature

Likelihood Of Impact: Very Low

`Illuminate\Database\Schema\Blueprint` constructor now expects an instance of `Illuminate\Database\Connection` as first argument.

Eloquent

Models and UUIDv7

Likelihood Of Impact: Medium

The `HasUuids` trait now generates UUIDs compatible with version 7 (ordered UUIDs). To use ordered UUIDv4, use `HasVersion4Uuids` instead:

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;

// Previous
use Illuminate\Database\Eloquent\Concerns\HasUuids;

// New for UUIDv4
use Illuminate\Database\Eloquent\Concerns\HasVersion4Uuids as HasUuids;
```

`HasVersion7Uuids` trait has been removed; use `HasUuids` for same behavior.

Requests

Nested Array Request Merging

Likelihood Of Impact: Low

`$request->mergeIfMissing()` now supports merging nested arrays with "dot" notation:

```
$request->mergeIfMissing([
    'user.last_name' => 'Otwell',
]);
```

Validation

Image Validation Now Excludes SVGs

The `image` rule no longer accepts SVGs by default. To permit SVGs:

```
use Illuminate\Validation\Rules\File;

'photo' => 'required|image:allow_svg'

// Or...
'photo' => [ 'required', File::image(allowSvg: true) ],
```

Miscellaneous

Review changes in the laravel/laravel GitHub repo. Many updates are optional but recommended for sync. Use the GitHub comparison tool to identify relevant updates.

Contribution Guide

Table of Contents

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 (Build Scripts)
- Laravel Horizon
- Laravel Livewire Starter Kit
- Laravel Passport
- Laravel Pennant
- Laravel Pint
- Laravel Prompts
- Laravel React Starter Kit

- Laravel Reverb
- Laravel Sail
- Laravel Sanctum
- Laravel Scout
- Laravel Socialite
- Laravel Telescope
- Laravel Vue Starter Kit

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 `12.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 `12.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 `[email protected]`. 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 (email protected).

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.

Installation - Laravel 12.x - The PHP Framework For Web Artisans

Main Content

- ```
                    Home
                </a>
            </li>
        </ul>

            ⌘K

        ```
        ```

```
            <div>
                <div>
```

```
    </div>

    <ul>
```

-

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
- Concurrency
- Context
- 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
- MongoDB

-

Eloquent ORM

- Getting Started
- Relationships
- Collections
- Mutators / Casts
- API Resources
- Serialization
- Factories

-

Testing

- Getting Started
- HTTP Tests
- Console Tests
- Browser Tests
- Database
- Mocking

-

Packages

- Cashier (Stripe)
- Cashier (Paddle)
- Dusk
- Envoy
- Fortify
- Folio
- Homestead
- Horizon
- Mix
- Octane
- Passport
- Pennant
- Pint
- Precognition
- Prompts
- Pulse

- Reverb
- Sail
- Sanctum
- Scout
- Socialite
- Telescope
- Valet

- API Documentation

```
</div>

            <div>
    <a href="#main-content">
Skip to content

    <div>
<aside>
            <div>
                <div>
                    <nav>
                        <div>
                            <ul>
```

-

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
- Concurrency
- Context
- 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
- MongoDB

Eloquent ORM

- Getting Started
- Relationships
- Collections
- Mutators / Casts
- API Resources
- Serialization
- Factories

Testing

- Getting Started
- HTTP Tests
- Console Tests
- Browser Tests
- Database
- Mocking

Packages

- Cashier (Stripe)
- Cashier (Paddle)
- Dusk
- Envoy
- Fortify
- Folio
- Homestead
- Horizon
- Mix
- Octane
- Passport
- Pennant
- Pint
- Precognition
- Prompts
- Pulse
- Reverb
- Sail
- Sanctum
- Scout
- Socialite
- Telescope
- Valet

- API Documentation

```
                        </div>
                    </nav>
                </div>
            </div>
        </aside>

        <section>
            <div>
                <section>
                    <section>


                        <div>
<h1>Installation</h1>
```


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.

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 Cloud allow you to run your Laravel application at nearly limitless scale.

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 Application

Installing PHP and the Laravel Installer

Before creating your first Laravel application, make sure that your local machine has PHP, Composer, and the Laravel installer installed. In addition, you should install either Node and NPM or Bun so that you can compile your application's frontend assets.

If you don't have PHP and Composer installed on your local machine, the following commands will install PHP, Composer, and the Laravel installer on macOS, Windows, or Linux:

macOS Windows PowerShell Linux

```
1/bin/bash -c "$(curl -fsSL https://php.new/install/mac/8.4)"

/bin/bash -c "$(curl -fsSL https://php.new/install/mac/8.4)"

1# Run as administrator...
2Set-ExecutionPolicy Bypass -Scope  Process -Force; [System.Net.ServicePointManager]::

# Run as administrator...
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::Se

1/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.4)"

/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.4)"
```

After running one of the commands above, you should restart your terminal session. To update PHP, Composer, and the Laravel installer after installing them via `php.new`, you can re-run the command in your terminal.

If you already have PHP and Composer installed, you may install the Laravel installer via Composer:

```
1composer global require laravel/installer

composer global require laravel/installer

</div>
<p>
```

For a fully-featured, graphical PHP installation and management experience, check out Laravel Herd.

Creating an Application

After you have installed PHP, Composer, and the Laravel installer, you're ready to create a new Laravel application. The Laravel installer will prompt you to select your preferred testing framework, database, and starter kit:

```
1laravel new example-app

laravel new example-app
```

Once the application has been created, you can start Laravel's local development server, queue worker, and Vite development server using the `dev` Composer script:

```
1cd example-app
2npm install && npm run build
3composer run dev

cd example-app
npm install && npm run build
composer run dev
```

Once you have started the 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.

```
</div>
<p>
```

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 `url` 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 be exposed.

```
</div>
<p>
```

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 an SQLite database.

During the creation of the application, Laravel created a `database/database.sqlite` file for you, and ran the necessary migrations to create the application's database tables.

If you prefer to use another database driver such as MySQL or PostgreSQL, you can update your `.env` configuration file to use the appropriate database. For example, if you wish to use MySQL, update your `.env` configuration file's `DB_*` variables like so:

```
1DB_CONNECTION=mysql
2DB_HOST=127.0.0.1
3DB_PORT=3306
4DB_DATABASE=laravel
5DB_USERNAME=root
6DB_PASSWORD=

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

If you choose to use a database other than SQLite, you will need to create the database and run your application's database migrations:

```
1php artisan migrate

php artisan migrate
```

```
</div>
<p>
```

If you are developing on macOS or Windows and need to install MySQL, PostgreSQL, or Redis locally, consider using Herd Pro or DBngin.

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.

Installation Using Herd

Laravel Herd is a blazing fast, native Laravel and PHP development environment for macOS and Windows. Herd includes everything you need to get started with Laravel development, including PHP and Nginx.

Once you install Herd, you're ready to start developing with Laravel. Herd includes command line tools for `php`, `composer`, `laravel`, `expose`, `node`, `npm`, and `nvm`.

```
</div>
<p>
```

Herd Pro augments Herd with additional powerful features, such as the ability to create and manage local MySQL, Postgres, and Redis databases, as well as local mail viewing and log monitoring.

Herd on macOS

If you develop on macOS, you can download the Herd installer from the Herd website. The installer automatically downloads the latest version of PHP and configures your Mac to always run Nginx in the background.

Herd for macOS uses dnsmasq to support "parked" directories. Any Laravel application in a parked directory will automatically be served by Herd. By default, Herd creates a parked directory at `~/Herd` and you can access any Laravel application in this directory on the `.test` domain using its directory name.

After installing Herd, the fastest way to create a new Laravel application is using the Laravel CLI, which is bundled with Herd:

```
1cd ~/Herd
2laravel new my-app
3cd my-app
4herd open
```

```
cd ~/Herd
laravel new my-app
cd my-app
herd open
```

Of course, you can always manage your parked directories and other PHP settings via Herd's UI, which can be opened from the Herd menu in your system tray.

You can learn more about Herd by checking out the Herd documentation.

Herd on Windows

You can download the Windows installer for Herd on the Herd website. After the installation finishes, you can start Herd to complete the onboarding process and access the Herd UI for the first time.

The Herd UI is accessible by left-clicking on Herd's system tray icon. A right-click opens the quick menu with access to all tools that you need on a daily basis.

During installation, Herd creates a "parked" directory in your home directory at `%USERPROFILE%\Herd`. Any Laravel application in a parked directory will automatically be served by Herd, and you can access any Laravel application in this directory on the `.test` domain using its directory name.

After installing Herd, the fastest way to create a new Laravel application is using the Laravel CLI, which is bundled with Herd. To get started, open Powershell and run the following commands:

```
1cd ~\Herd
2laravel new my-app
3cd my-app
4herd open
```

```
cd ~\Herd
laravel new my-app
cd my-app
herd open
```

You can learn more about Herd by checking out the Herd documentation for Windows.

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.

If you develop in Visual Studio Code (VS Code), the official Laravel VS Code Extension is now available. This extension brings Laravel-specific tools directly into your VS Code environment, enhancing productivity.

Next Steps

Now that you have created your Laravel application, 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.

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.

```
</div>
<p>
```

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.

```
        <div>
            <div>
                <div>
<h3>
    On this page
</h3>

<div>
    <div>
                        <ul>
                                <li>
                        <a href="#meet-laravel">
                            Meet Laravel
                        </a>

                                                <ul>
                                                                <li>
                                <a href="#why-laravel">
                                    Why Laravel?
                                </a>
                            </li>
                                                        </ul>
                                    </li>
                                <li>
                        <a href="#creating-a-laravel-project">
                            Creating a Laravel Application
                        </a>

                                                <ul>
                                                                <li>
                                <a href="#installing-php">
                                    Installing PHP and the Laravel Installer
                                </a>
                            </li>
                                                                <li>
                                <a href="#creating-an-application">
                                    Creating an Application
                                </a>
                            </li>
                                                        </ul>
                                    </li>
                                <li>
                        <a href="#initial-configuration">
                            Initial Configuration
                        </a>

                                                <ul>
                                                                <li>
                                <a href="#environment-based-configuration">
                                    Environment Based Configuration
                                </a>
                            </li>
                                                                <li>
                                <a href="#databases-and-migrations">
                                    Databases and Migrations
                                </a>
                            </li>
                                                                <li>
```

```
                                    <a href="#directory-configuration">
                                        Directory Configuration
                                    </a>
                                </li>
                                                                    </ul>
                                        </li>
                                    <li>
                        <a href="#installation-using-herd">
                            Installation Using Herd
                        </a>

                                                <ul>
                                                                    <li>
                                    <a href="#herd-on-macos">
                                        Herd on macOS
                                    </a>
                                </li>
                                                                    <li>
                                    <a href="#herd-on-windows">
                                        Herd on Windows
                                    </a>
                                </li>
                                                                    </ul>
                                        </li>
                                    <li>
                        <a href="#ide-support">
                            IDE Support
                        </a>

                                        </li>
                                    <li>
                        <a href="#next-steps">
                            Next Steps
                        </a>

                                                <ul>
                                                                    <li>
                                    <a href="#laravel-the-fullstack-framework">
                                        Laravel the Full Stack Framework
                                    </a>
                                </li>
                                                                    <li>
                                    <a href="#laravel-the-api-backend">
                                        Laravel the API Backend
                                    </a>
                                </li>
                                                                    </ul>
                                        </li>
                                </ul>
                </div>
    </div>

                </div>
            </div>

    <div>

    <div>
```

```
    <a href="#">


    </a>

    <a href="https://laracon.us" target="_blank">

    </a>
</div>

        </div>
    </div>

    <div>
<div>
    <div>
        <div>
            <p>Laravel is the most productive way to<br /> build, deploy, and monitor

            <ul>
                <li>
                    <a href="https://github.com/laravel" target="_blank">


                    </a>
                </li>
                <li>
                    <a href="https://x.com/laravelphp" target="_blank">


                    </a>
                </li>
                <li>
                    <a href="https://www.youtube.com/@LaravelPHP" target="_blank">



                    </a>
                </li>
                <li>
                    <a href="https://discord.com/invite/laravel" target="_blank">


                    </a>
                </li>
            </ul>

            <div>
                <ul>
                    <li>© 2025 Laravel</li>
                    <li><a href="https://laravel.com/legal">Legal</a></li>
                    <li><a href="https://status.laravel.com/">Status</a></li>
                </ul>
            </div>
        </div>
```

```
        <div>
<div>
<h4>
                        Products
                    </h4>

                                                <ul>
                                                                    <li>
                                    <a href="https://cloud.laravel.com">Cloud</a>
                                </li>
                                                                    <li>
                                    <a href="https://forge.laravel.com">Forge</a>
                                </li>
                                                                    <li>
                                    <a href="https://nightwatch.laravel.com">Nightwatc
                                </li>
                                                                    <li>
                                    <a href="https://vapor.laravel.com">Vapor</a>
                                </li>
                                                                    <li>
                                    <a href="https://nova.laravel.com">Nova</a>
                                </li>
                                                            </ul>
```

Packages

```
                                                <ul>
                                                                    <li>
                                    <a href="/docs/cashier">Cashier</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/dusk">Dusk</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/horizon">Horizon</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/octane">Octane</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/scout">Scout</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/pennant">Pennant</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/pint">Pint</a>
                                </li>
                                                                    <li>
                                    <a href="/docs/sail">Sail</a>
                                </li>
                                                                    <li>
```

```
                    <a href="/docs/sanctum">Sanctum</a>
                </li>
                                        <li>
                    <a href="/docs/socialite">Socialite</a>
                </li>
                                        <li>
                    <a href="/docs/telescope">Telescope</a>
                </li>
                                        <li>
                    <a href="/docs/pulse">Pulse</a>
                </li>
                                        <li>
                    <a href="/docs/reverb">Reverb</a>
                </li>
                                        <li>
                    <a href="/docs/broadcasting">Echo</a>
                </li>
                                </ul>
```

Resources

```
                        <ul>
                                        <li>
                    <a href="/docs">Documentation</a>
                </li>
                                        <li>
                    <a href="/starter-kits">Starter Kits</a>
                </li>
                                        <li>
                    <a href="/docs/releases">Release Notes</a>
                </li>
                                        <li>
                    <a href="https://blog.laravel.com">Blog</a>
                </li>
                                        <li>
                    <a href="https://laravel-news.com">News</a>
                </li>
                                        <li>
                    <a href="https://larabelles.com/">Larabelles</a>
                </li>
                                        <li>
                    <a href="https://larajobs.com/?partner=5">Jobs</a>
                </li>
                                        <li>
                    <a href="/careers">Careers</a>
                </li>
                                        <li>
                    <a href="https://trust.laravel.com">Trust</a></li>
            </ul>
```

Partners

```
                        <ul>
                                        <li>
                    <a href="https://partners.laravel.com/partners/red
                </li>
                                        <li>
                    <a href="https://partners.laravel.com/partners/veh
                </li>
```

```
                                    <li>
                                <a href="https://partners.laravel.com/partners/kir
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com/partners/act
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com/partners/cur
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com/partners/dev
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com/partners/tig
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com/partners/64-
                            </li>
                                    <li>
                                <a href="https://partners.laravel.com">More Partne
                            </li>
                                </ul>

        <div>


</div>
    </div>
</div>
```

Configuration

Introduction

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

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

The `about` Command

Laravel can display an 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 read by the configuration files within the `config` directory using Laravel's `env` function.

If you are developing with a team, you may wish to continue including and updating the `.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 if an intruder gains access to your source control repository, as any sensitive credentials would be 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:

`.env` Value	`env()` Return
true	`(bool) true`
(true)	`(bool) true`
false	`(bool) false`
(false)	`(bool) false`
empty	`(string) ''`
(empty)	`(string) ''`
null	`(null) null`
(null)	`(null) null`

If you need to define an environment variable with a value that contains spaces, enclose the value in double quotes:

```
APP_NAME="My Application"
```

Retrieving Environment Configuration

All variables listed in the `.env` file are loaded into the `$_ENV` PHP super-global when your application receives a request. You may use the `env` function to retrieve these values in your configuration files. Many of Laravel's configuration options use this function internally:

```
'default' => env('APP_DEBUG', false),
```

The second value is the default, returned if the environment variable does not exist.

Determining the Current Environment

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

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

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

You can pass arguments to this method to check if the environment matches a given value:

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

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

The environment detection can be overridden by defining a server-level `APP_ENV` variable.

Encrypting Environment Files

Unencrypted environment files should never be stored in source control. Laravel allows you to encrypt your environment files so they can be safely added to source control.

Encryption

To encrypt an environment file, use:

```
php artisan env:encrypt
```

This encrypts `.env` and creates `.env.encrypted`. The decryption key is output and should be stored securely. To specify your own key:

```
php artisan env:encrypt --key=your-custom-key
```

The key length must match the encryption cipher used. Default cipher is `AES-256-CBC`, requiring a 32-character key. You can specify a cipher with `--cipher`.

For multiple environment files:

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

Decryption

To decrypt:

```
php artisan env:decrypt
```

Or specify a key:

```
php artisan env:decrypt --key=your-custom-key
```

You can specify the environment file with `--env`. To overwrite existing `.env`:

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

Accessing Configuration Values

Access configuration values with the `Config` facade or `config()` helper:

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

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

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

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

Set values at runtime:

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

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

`Config` facade provides typed retrieval methods:

```
Config::string('config-key');
Config::integer('config-key');
Config::float('config-key');
Config::boolean('config-key');
Config::array('config-key');
Config::collection('config-key');
```

Configuration Caching

To improve performance, cache all configuration into a single file:

```
php artisan config:cache
```

Avoid running during local development. Once cached, the `.env` file is not loaded in requests; `env()` returns external environment variables only.

Clear the cache with:

```
php artisan config:clear
```

Configuration Publishing

Laravel's default configuration files are already published. To publish additional configuration files:

```
php artisan config:publish
```

With `--all`:

```
php artisan config:publish --all
```

Debug Mode

In `config/app.php`, the `debug` option determines error details shown. Defaults to the `APP_DEBUG` environment variable.

Set `APP_DEBUG=true` during local development. In production, always set to `false`. Exposing debug info in production entails security risks.

Maintenance Mode

Put your application in maintenance mode to display a custom view:

```
php artisan down
```

Options include:

- `--refresh=seconds` to auto-refresh
- `--retry=seconds` to set `Retry-After` header
- `--secret=<token>` to allow bypass with secret token
- `--with-secret` to generate a token

- `--render=<view>` to pre-render the view
- `--redirect=<uri>` to redirect all requests

Disable maintenance mode:

```
php artisan up
```

Maintenance Mode and Queues

While in maintenance mode, queued jobs are not processed. Once out of maintenance, jobs resume.

Alternatives to Maintenance Mode

For zero-downtime deployments, consider managed platforms like Laravel Cloud.

(Additional content like footer, social links, and extra pages follow in similar structured markdown.)

Directory Structure

Introduction

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

The Root Directory

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.

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, two route files are included with Laravel: `web.php` and `console.php`.

- The `web.php` file contains routes that Laravel places in the `web` middleware group, which provides session state, CSRF protection, and cookie encryption.

- The `console.php` file is where you may define all of your closure-based console commands. It does not define HTTP routes but allows defining console-based entry points.

You may also install additional route files for API routes (`api.php`) and broadcasting channels (`channels.php`) via artisan commands.

The Storage Directory

The `storage` directory contains your logs, compiled Blade templates, file-based sessions, file caches, and other files generated by the framework. It includes subdirectories:

- `app` : for files generated by your application.
- `framework` : for framework generated files and caches.
- `logs` : for your application's log files.

The `storage/app/public` directory may be used to store user-generated files, such as profile avatars. You should create a symbolic link at `public/storage` pointing to this directory using the command:

```
php artisan storage:link
```

The Tests Directory

The `tests` directory contains your automated tests, including examples using Pest or PHPUnit. Test classes should be suffixed with `Test` . You can run tests with:

- `/vendor/bin/pest`
- `/vendor/bin/phpunit`
- `php artisan test`

The Vendor Directory

The `vendor` directory contains your Composer dependencies.

The App Directory (Detailed)

The majority of your application code is housed here. It's namespaced under `App` and autoloaded via PSR-4.

It contains subdirectories like:

- `Http` : controllers, middleware, form requests.
- `Models` : your Eloquent ORM models.
- `Providers` : service providers.
- `Console` , `Events` , `Exceptions` , `Jobs` , `Listeners` , `Mail` , `Notifications` , `Policies` , `Rules` , and more.

Many classes can be generated via artisan commands such as `php artisan make:model` , `make:controller` , etc.

The Broadcasting Directory

Contains broadcast channel classes, generated with `make:channel` .

The Console Directory

Contains custom artisan commands, generated with `make:command` .

The Events Directory

Houses event classes, generated with `event:generate` or `make:event`.

The Exceptions Directory

Contains custom exception classes, generated with `make:exception`.

The Http Directory

Contains controllers, middleware, and request classes.

The Jobs Directory

Houses queueable jobs, created with `make:job`.

The Listeners Directory

Contains classes handling events, generated with `make:listener`.

The Mail Directory

Contains email classes, created with `make:mail`.

The Models Directory

Contains your Eloquent models.

The Notifications Directory

Houses notification classes, generated with `make:notification`.

The Policies Directory

Contains authorization policy classes, generated with `make:policy`.

The Providers Directory

Contains service providers, including the default `AppServiceProvider`.

The Rules Directory

Contains custom validation rule classes, created with `make:rule`.

Hello

Hello

Deployment

Introduction

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

Server Requirements

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

- PHP >= 8.2
- 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 fully-managed Laravel platform like Laravel Cloud.**

Please ensure 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 ~ ^/index\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
        fastcgi_hide_header X-Powered-By;
    }

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

FrankenPHP

FrankenPHP may also be used to serve your Laravel applications. FrankenPHP is a modern PHP application server written in Go. To serve a Laravel PHP application using FrankenPHP, you may simply invoke its `php-server` command:

```
frankenphp php-server -r public/
```

To take advantage of more powerful features supported by FrankenPHP, such as its Laravel Octane integration, HTTP/3, modern compression, or the ability to package Laravel applications as standalone binaries, please consult FrankenPHP's Laravel documentation.

Directory Permissions

Laravel will need to write to the `bootstrap/cache` and `storage` directories. Ensure that the web server process owner has permission to write to these directories.

Optimization

When deploying your application to production, it is essential to cache certain files such as configuration, events, routes, and views. Laravel provides an `artisan` command `optimize` to cache all of these files, typically invoked during deployment:

```
php artisan optimize
```

To remove all cache files generated by `optimize` and clear cache keys:

```
php artisan optimize:clear
```

Caching Configuration

Run the following command during deployment:

```
php artisan config:cache
```

This will combine all configuration files into a single cached file, reducing filesystem trips when loading configuration values. Be sure to only call `env` from configuration files, as environment variables are not available once cache is compiled:

> **Note:** Once cached, the `.env` file will not be loaded, and calls to `env()` will return `null`.

Caching Events

Cache your application's auto-discovered event-to-listener mappings:

```
php artisan event:cache
```

Caching Routes

For large applications with many routes, cache route registrations:

```
php artisan route:cache
```

Caching Views

Precompile Blade views to improve performance:

```
php artisan view:cache
```

Debug Mode

The `debug` option in `config/app.php` determines how much error information is displayed. It defaults to the `APP_DEBUG` environment variable stored in `.env`. **In production, this should always be set to `false` to avoid exposing sensitive information.**

The Health Route

Laravel includes a built-in health check route, useful for monitoring application status in production. It is typically served at `/up` or `/status`, returning 200 if healthy, or 500 if not.

Configure the URI in `bootstrap/app`:

```
->withRouting([
    'web' => __DIR__.'/../routes/web.php',
    'commands' => __DIR__.'/../routes/console.php',
    'health' => '/up',
])
```

Requests to this route dispatch an `Illuminate\Foundation\Events\DiagnosingHealth` event, allowing custom health checks through event listeners.

Deploying With Laravel Cloud or Forge

Laravel Cloud

If you want a fully-managed, scalable deployment platform, check out Laravel Cloud. It offers managed compute, databases, caches, and object storage, tailored for Laravel apps.

Laravel Forge

If you manage your own servers but want simplified server management, Laravel Forge supports creating and managing servers on providers like DigitalOcean, Linode, AWS, etc. Forge installs and manages PHP, web server, database, Redis, and other tools needed for Laravel applications.

Request Lifecycle

Introduction

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

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

Lifecycle Overview

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, using the `handleRequest` or `handleCommand` methods of the application instance, depending on the type of request entering the application. These two kernels serve as the central location through which all requests flow. For now, let's just focus on the HTTP kernel, which is an instance of `Illuminate\Foundation\Http\Kernel`.

The HTTP kernel 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 is also responsible for passing the request through the application's middleware stack. 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.

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.

While the framework internally uses dozens of service providers, you also have the option to create your own. You can find a list of the user-defined or third-party service providers that your application is using in the `bootstrap/providers.php` file.

Routing

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 `PreventRequestsDuringMaintenance`, 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 to the `handleRequest` of the application instance, and this method calls the `send` method on the returned response. The `send` method sends the response content to the user's web browser. We've now completed 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 user-defined 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

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\Services\AppleMusic;
use Illuminate\View\View;

class PodcastController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected AppleMusic $apple,
    ) {}

    /**
     * Show information about the given podcast.
     */
    public function show(string $id): View
    {
        return view('podcasts.show', [
            'podcast' => $this->apple->findPodcast($id)
        ]);
    }
}
```

In this example, the `PodcastController` needs to retrieve podcasts from a data source such as Apple Music. So, we will **inject** a service that is able to retrieve podcasts. Since the service is injected, we are able to easily *mock*, or create a dummy implementation of the `AppleMusic` service 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 $service) {
    dd($service::class);
});
```

The container will automatically resolve `Service` without additional configuration.

Binding

Binding Basics

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. You can register a binding like this:

```
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, which we can then use to resolve sub-dependencies.

Interacting Outside Service Providers

You can also interact with the container outside of service providers 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) {
    // ...
});
```

Conditional Binding

You may use `bindIf` to register a binding only if it has not already been registered:

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

Inferring Class Types

Laravel can infer the class type from the return type of the closure:

```
App::bind(function (Application $app): Transistor {
    return new Transistor($app->make(PodcastParser::class));
```

```
});
```

Binding A Singleton

The `singleton` method binds a class or interface into the container that should only be resolved once. Subsequent calls return the same instance:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

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

You can also conditionally bind singletons with `singletonIf`:

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

Binding Scoped Singletons

The `scoped` method binds instances only within a single request or job lifecycle, and they are flushed when the lifecycle resets:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

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

Similarly, `scopedIf` registers the scoped binding only if not already registered:

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

Binding Instances

You can bind an existing object instance directly:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);
$this->app->instance(Transistor::class, $service);
```

Binding Interfaces to Implementations

Bind an interface to a specific implementation to facilitate dependency injection:

```
use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

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

Now, when you type-hint the `EventPusher` interface, Laravel injects `RedisEventPusher`.

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 */
public function __construct(
    protected EventPusher $pusher,
) {}
```

Contextual Binding

Define different implementations for interfaces depending on the class requesting them:

```
use App\Http\Controllers\PhotoController;
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');
    });
```

Contextual Attributes

Inject specific values or configuration directly into classes:

```
namespace App\Http\Controllers;

use Illuminate\Container\Attributes\Storage;
use Illuminate\Contracts\Filesystem\Filesystem;

class PhotoController extends Controller
{
    public function __construct(
        #[Storage('local')] protected Filesystem $filesystem
    ) {
        // ...
    }
}
```

Additional attributes include `Auth`, `Cache`, `Config`, `Context`, `DB`, `Give`, `Log`, `RouteParameter`, `Tag`, and more, allowing fine-grained dependency management.

```
namespace App\Http\Controllers;

use App\Contracts\UserRepository;
```

```
use App\Models\Photo;
use App\Repositories\DatabaseRepository;
use Illuminate\Container\Attributes\Auth;
use Illuminate\Container\Attributes\Cache;
use Illuminate\Container\Attributes\Config;
use Illuminate\Container\Attributes\Context;
use Illuminate\Container\Attributes\DB;
use Illuminate\Container\Attributes\Give;
use Illuminate\Container\Attributes\Log;
use Illuminate\Container\Attributes\RouteParameter;
use Illuminate\Container\Attributes\Tag;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Database\Connection;
use Psr\Log\LoggerInterface;

class PhotoController extends Controller
{
    public function __construct(
        #[Auth('web')] protected Guard $auth,
        #[Cache('redis')] protected Repository $cache,
        #[Config('app.timezone')] protected string $timezone,
        #[Context('uuid')] protected string $uuid,
        #[Context('ulid', hidden: true)] protected string $ulid,
        #[DB('mysql')] protected Connection $connection,
        #[Give(DatabaseRepository::class)] protected UserRepository $users,
        #[Log('daily')] protected LoggerInterface $log,
        #[RouteParameter('photo')] protected Photo $photo,
        #[Tag('reports')] protected iterable $reports,
    ) {
        // ...
    }
}
```

Laravel also provides a `CurrentUser` attribute to inject the authenticated user:

```
use App\Models\User;
use Illuminate\Container\Attributes\CurrentUser;

Route::get('/user', function (#[CurrentUser] User $user) {
    return $user;
})->middleware('auth');
```

Defining Custom Attributes

Create your own contextual attributes by implementing `Illuminate\Contracts\Container\ContextualAttribute`. Example: re-implementing Laravel's `Config` attribute:

```
<?php
namespace App\Attributes;

use Attribute;
use Illuminate\Contracts\Container\Container;
use Illuminate\Contracts\Container\ContextualAttribute;
```

```
#[Attribute(Attribute::TARGET_PARAMETER)]
class Config implements ContextualAttribute
{
    public function __construct(public string $key, public mixed $default = null) {}

    public static function resolve(self $attribute, Container $container)
    {
        return $container->make('config')->get($attribute->key, $attribute->default);
    }
}
```

Binding Primitives

Inject primitive values such as integers or strings:

```
use App\Http\Controllers\UserController;

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

Inject tagged dependencies:

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

Inject configuration values:

```
$this->app->when(ReportAggregator::class)
    ->needs('$timezone')
    ->giveConfig('app.timezone');
```

Binding Typed Variadics

Inject arrays of typed objects via variadic constructor parameters:

```
<?php
use App\Models\Filter;
use App\Services\Logger;

class Firewall
{
    /**
     * The filter instances.
     *
     * @var array
     */
    protected $filters;

    public function __construct(
        protected Logger $logger,
        Filter ...$filters,
    ) {
        $this->filters = $filters;
```

```
    }
}
```

Resolve with a closure returning an array:

```
$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),
        ];
    });
```

Or provide an array of class names:

```
$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give([
        NullFilter::class,
        ProfanityFilter::class,
        TooLongFilter::class,
    ]);
```

Variadic Tag Dependencies

Inject all tagged implementations of a class:

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

Tagging

Assign a tag to multiple bindings:

```
$this->app->bind(CpuReport::class, function () {
    // ...
});
$this->app->bind(MemoryReport::class, function () {
    // ...
});
$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');
```

Resolve all services with a tag:

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

Extending Bindings

Modify already resolved services:

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

Resolving

The `make` Method

Resolve a class instance:

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

The `makeWith` Method

Pass additional constructor parameters:

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

Check if a Binding Exists

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

Outside Helper or Facade

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

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

Injecting the Container

Type-hint `Illuminate\Container\Container` in constructor:

```
use Illuminate\Container\Container;

public function __construct(
    protected Container $container,
) {}
```

Automatic Injection

Type-hint dependencies in constructor, method, or job handle:

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

use App\Services\AppleMusic;

class PodcastController extends Controller
{
```

```
    public function __construct(
        protected AppleMusic $apple,
    ) {}

    public function show(string $id): Podcast
    {
        return $this->apple->findPodcast($id);
    }
}
```

Or in method parameters:

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

use App\Services\AppleMusic;

class PodcastController extends Controller
{
    public function show(AppleMusic $apple, string $id): Podcast
    {
        return $apple->findPodcast($id);
    }
}
```

Method Invocation and Injection

Invoke methods on objects with dependencies injected:

```
<?php
namespace App;

use App\Services\AppleMusic;

class PodcastStats
{
    /**
     * Generate a new podcast stats report.
     */
    public function generate(AppleMusic $apple): array
    {
        return [
            // ...
        ];
    }
}
```

Call methods via container:

```
use App\PodcastStats;

$stats = App::call([new PodcastStats, 'generate']);
```

Or invoke closures with dependencies:

```
use App\Services\AppleMusic;
```

```
$result = App::call(function (AppleMusic $apple) {
    // ...
});
```

Container Events

Listen to resolving events:

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

Rebinding events:

```
$this->app->rebinding(PodcastPublisher::class, function (Application $app, PodcastPubl
    // ...
});
// To override existing binding:
$this->app->bind(PodcastPublisher::class, TransistorPublisher::class);
```

PSR-11 Interface

Laravel's container implements PSR-11:

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);
    // ...
});
```

Exceptions will be of type `Psr\Container\NotFoundExceptionInterface` or `Psr\Container\ContainerExceptionInterface` depending on the failure.

Service 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.

Laravel uses dozens of service providers internally to bootstrap its core services, 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.

All user-defined service providers are registered in the `bootstrap/providers.php` file. In the following documentation, 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. Laravel will automatically register your new provider in your application's `bootstrap/providers.php` file:

```
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\Applicaton;
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\ServerProvider;
use App\Services\DigitalOceanServerProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * All of the container bindings that should be registered.
     *
     * @var array
     */
    public $bindings = [
        ServerProvider::class => DigitalOceanServerProvider::class,
    ];

    /**
     * All of the container singletons that should be registered.
     *
     * @var array
     */
    public $singletons = [
        DowntimeNotifier::class => PingdomDowntimeNotifier::class,
        ServerProvider::class => ServerToolsProvider::class,
    ];
}
```

The Boot Method

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:

```
<?php

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 `bootstrap/providers.php` configuration file. This file returns an array that contains the class names of your application's service providers:

```
<?php

return [
    App\Providers\AppServiceProvider::class,
];
```

When you invoke the `make:provider` Artisan command, Laravel will automatically add the generated provider to the `bootstrap/providers.php` file. However, if you have manually created the provider class, you should manually add the provider class to the array:

```
<?php

return [
    App\Providers\AppServiceProvider::class,
    App\Providers\ComposerServiceProvider::class,
];
```

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

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

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([
        // ...
    ]);
});
```

or

```
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;

test('basic example', function () {
    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
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function showProfile(string $id): \Illuminate\View\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`:

```php
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
<?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
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
        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:

```
use App\Models\Podcast;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('podcast can be published', function () {
    $podcast = Podcast::factory()->create();

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

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

or in a feature test:

```
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;

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

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

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

When the facade is invoked, it will resolve the underlying class from the service container, allowing you to call methods directly as if it was a static class.

Facade Class Reference

Below you'll 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 (Instance)	Illuminate\Contracts\Auth\Guard	`auth.driver`
Auth	Illuminate\Auth\AuthManager	`auth`
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Broadcasting (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Broadcasting	Illuminate\Contracts\Broadcasting\Factory	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache (Instance)	Illuminate\Cache\Repository	`cache.store`
Cache	Illuminate\Cache\CacheManager	`cache`
Config	Illuminate\Config\Repository	`config`
Context	Illuminate\Log\Context\Repository	
Cookie	Illuminate\Cookie\CookieJar	`cookie`
Crypt	Illuminate\Encryption\Encrypter	`encrypter`
Date	Illuminate\Support\DateFactory	`date`
DB (Instance)	Illuminate\Database\Connection	`db.connection`
DB	Illuminate\Database\DatabaseManager	`db`

Facade	Class	Service Container Binding
Event	Illuminate\Events\Dispatcher	`events`
Exceptions (Instance)	Illuminate\Contracts\Debug\ExceptionHandler	
Exceptions	Illuminate\Foundation\Exceptions\Handler	
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 (Instance)	Illuminate\Auth\Passwords\PasswordBroker	`auth.password.broker`
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	`auth.password`
Pipeline (Instance)	Illuminate\Pipeline\Pipeline	
Process	Illuminate\Process\Factory	
Queue (Base Class)	Illuminate\Queue\Queue	
Queue (Instance)	Illuminate\Contracts\Queue\Queue	`queue.connection`
Queue	Illuminate\Queue\QueueManager	`queue`
RateLimiter	Illuminate\Cache\RateLimiter	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis (Instance)	Illuminate\Redis\Connections\Connection	`redis.connection`
Redis	Illuminate\Redis\RedisManager	`redis`
Request	Illuminate\Http\Request	`request`
Response (Instance)	Illuminate\Http\Response	
Response	Illuminate\Contracts\Routing\ResponseFactory	
Route	Illuminate\Routing\Router	`router`
Schedule	Illuminate\Console\Scheduling\Schedule	
Schema	Illuminate\Database\Schema\Builder	
Session (Instance)	Illuminate\Session\Store	`session.store`
Session	Illuminate\Session\SessionManager	`session`
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	`filesystem.disk`
Storage	Illuminate\Filesystem\FilesystemManager	`filesystem`
URL	Illuminate\Routing\UrlGenerator	`url`
Validator (Instance)	Illuminate\Validation\Validator	
Validator	Illuminate\Validation\Factory	`validator`
View (Instance)	Illuminate\View\View	
View	Illuminate\View\Factory	`view`
Vite	Illuminate\Foundation\Vite	

Hello

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. A variety of middleware are included in Laravel, including middleware for authentication and CSRF protection; however, all user-defined middleware are typically located in your application's `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\
     */
    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, you may append it to the global middleware stack in your application's `bootstrap/app.php` file:

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

$app->middleware([
    EnsureTokenIsValid::class,
]);
```

Or using the `withMiddleware` method:

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

->withMiddleware(function (Middleware $middleware) {
    $middleware->append(EnsureTokenIsValid::class);
})
```

The `$middleware` object provided to the `withMiddleware` closure is an instance of `Illuminate\Foundation\Configuration\Middleware` and is responsible for managing the middleware assigned to your application's routes. The `append` method adds the middleware to the end of the list of global middleware. If you wish to add it at the beginning, use `prepend`.

Manually Managing Laravel's Default Global Middleware

You can specify Laravel's default middleware stack and adjust as necessary:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->use([
        \Illuminate\Foundation\Http\Middleware\InvokeDeferredCallbacks::class,
        // \Illuminate\Http\Middleware\TrustHosts::class,
        \Illuminate\Http\Middleware\TrustProxies::class,
        \Illuminate\Http\Middleware\HandleCors::class,
        \Illuminate\Foundation\Http\Middleware\PreventRequestsDuringMaintenance::class
        \Illuminate\Http\Middleware\ValidatePostSize::class,
        \Illuminate\Foundation\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    ]);
})
```

Assigning Middleware to Routes

To assign middleware to specific routes:

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

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

You can assign multiple middleware by passing an array:

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

Excluding Middleware

To prevent middleware from applying to a specific route within a group:

```
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 middleware from an entire route group:

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

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

The `withoutMiddleware` method only removes route middleware, not global middleware.

Middleware Groups

Group multiple middleware under a single key using `appendToGroup`:

```
use App\Http\Middleware\First;
use App\Http\Middleware\Second;

->withMiddleware(function (Middleware $middleware) {
    $middleware->appendToGroup('group-name', [
        First::class,
        Second::class,
    ]);
    $middleware->prependToGroup('group-name', [
        First::class,
        Second::class,
    ]);
});
```

Assign middleware groups to routes:

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

Route::middleware(['group-name'])->group(function () {
    // ...
});
```

Laravel's Default Middleware Groups

Laravel includes predefined `web` and `api` groups:

`web` Middleware Group
`Illuminate\Cookie\Middleware\EncryptCookies`
`Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse`
`Illuminate\Session\Middleware\StartSession`
`Illuminate\View\Middleware\ShareErrorsFromSession`
`Illuminate\Foundation\Http\Middleware\ValidateCsrfToken`
`Illuminate\Routing\Middleware\SubstituteBindings`

`api` Middleware Group
`Illuminate\Routing\Middleware\SubstituteBindings`

You can modify these groups using the `web` and `api` methods in `bootstrap/app.php`.

Middleware Aliases

Alias middleware for shorter names:

```
use App\Http\Middleware\EnsureUserIsSubscribed;

->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'subscribed' => EnsureUserIsSubscribed::class,
    ]);
});
```

Use the alias in routes:

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

Default Laravel aliases include:

Alias	Middleware
`auth`	`Illuminate\Auth\Middleware\Authenticate`
`auth.basic`	`Illuminate\Auth\Middleware\AuthenticateWithBasicAuth`
`auth.session`	`Illuminate\Session\Middleware\AuthenticateSession`
`cache.headers`	`Illuminate\Http\Middleware\SetCacheHeaders`
`can`	`Illuminate\Auth\Middleware\Authorize`
`guest`	`Illuminate\Auth\Middleware\RedirectIfAuthenticated`
`password.confirm`	`Illuminate\Auth\Middleware\RequirePassword`
`signed`	`Illuminate\Routing\Middleware\ValidateSignature`
`subscribed`	`\Spark\Http\Middleware\VerifyBillableIsSubscribed`
`throttle`	`Illuminate\Routing\Middleware\ThrottleRequests`
`verified`	`Illuminate\Auth\Middleware\EnsureEmailIsVerified`

Sorting Middleware

Specify execution order with the `priority` method:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->priority([
        \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
```

```
        // ...
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
        \Illuminate\Auth\Middleware\Authorize::class,
    ]);
})
```

Middleware Parameters

Middleware can accept additional parameters. Define them separated by `:` and multiple parameters by commas.

Example:

```
use App\Http\Middleware\EnsureUserHasRole;

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

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

Middleware receiving parameters:

```
<?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\
     */
    public function handle(Request $request, Closure $next, string $role): Response
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

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

Terminable Middleware

Middleware may have a `terminate` method for after-response tasks, automatically called when using FastCGI:

```php
<?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\
     */
    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
    {
        // ...
    }
}
```

Register it as singleton in your service provider:

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

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

HTTP Session - Laravel 12.x - The PHP Framework For Web Artisans

Hello

Error Handling

Introduction

When you start a new Laravel project, error and exception handling is already configured for you; however, at any point, you may use the `withExceptions` method in your application's `bootstrap/app.php` to manage how exceptions are reported and rendered by your application.

The `$exceptions` object provided to the `withExceptions` closure is an instance of `Illuminate\Foundation\Configuration\Exceptions` and is responsible for managing exception handling in your application. We'll dive deeper into this object 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.**

Handling Exceptions

Reporting Exceptions

In Laravel, exception reporting is used to log exceptions or send them to an external service like Sentry or Flare. 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 `report` exception method in your application's `bootstrap/app.php` 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;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->report(function (InvalidOrderException $e) {
        // ...
    });
});
```

When you register a custom exception reporting callback using the `report` 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:

```
use App\Exceptions\InvalidOrderException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->report(function (InvalidOrderException $e) {
```

```
        // ...
    })->stop();

    $exceptions->report(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 using the `context` exception method in your application's `bootstrap/app.php` file. This information will be included in every exception's log message written by your application:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->context(fn () => [
        '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:

```
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 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 invoke the `dontReportDuplicates` exception method in your application's `bootstrap/app.php` file:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->dontReportDuplicates();
});
```

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.

Even when you register a custom exception reporting callback using the `report` method, Laravel will still log the exception using the default logging configuration for the application. Since the log level can influence the channels on which a message is logged, you may wish to configure the log level at which certain exceptions are logged.

You can specify the log level for an exception type using the `level` exception method:

```
use PDOException;
use Psr\Log\LogLevel;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->level(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, you may use the `dontReport` exception method in your application's `bootstrap/app.php` file. Any class provided to this method will never be reported; however, they may still have custom rendering logic:

```
use App\Exceptions\InvalidOrderException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->dontReport([
        InvalidOrderException::class,
    ]);
});
```

Alternatively, you may simply "mark" an exception class with the `Illuminate\Contracts\Debug\ShouldntReport` interface. When an exception is marked with this interface, it will never be reported by Laravel's exception handler:

```
namespace App\Exceptions;

use Exception;
use Illuminate\Contracts\Debug\ShouldntReport;

class PodcastProcessingException extends Exception implements ShouldntReport
{
    // ...
}
```

Laravel already ignores certain errors internally, such as exceptions resulting from 404 HTTP errors or 419 HTTP responses due to invalid CSRF tokens. If you want Laravel to stop ignoring a specific exception type, you may use the `stopIgnoring` exception method:

```
use Symfony\Component\HttpKernel\Exception\HttpException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->stopIgnoring(HttpException::class);
});
```

Rendering Exceptions

By default, Laravel's exception handler converts exceptions into an HTTP response. You can register a custom rendering closure for exceptions of a specific type:

```
use App\Exceptions\InvalidOrderException;
use Illuminate\Http\Request;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->render(function (InvalidOrderException $e, Request $request) {
        return response()->view('errors.invalid-order', ['status' => 500]);
    });
});
```

You may also override rendering behavior for built-in Laravel or Symfony exceptions such as `NotFoundHttpException`. If the closure does not return a value, Laravel's default exception response is used:

```
use Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
```

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->render(function (NotFoundHttpException $e, Request $request) {
        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Record not found.',
            ], 404);
        }
    });
});
```

Rendering Exceptions as JSON

Laravel automatically determines whether to render exceptions as HTML or JSON based on the request's `Accept` header. To customize this behavior, you can use the `shouldRenderJsonWhen` method:

```
use Illuminate\Http\Request;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->shouldRenderJsonWhen(function (Request $request, Throwable $e) {
        if ($request->is('admin/*')) {
            return true;
        }
        return $request->expectsJson();
    });
});
```

Customizing the Exception Response

You can customize Laravel's entire exception response by registering a response closure with the `respond` method:

```
use Symfony\Component\HttpFoundation\Response;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->respond(function (Response $response) {
        if ($response->getStatusCode() === 419) {
            return back()->with([
                'message' => 'The page expired, please try again.',
            ]);
        }
        return $response;
    });
});
```

Reportable and Renderable Exceptions

Instead of defining behavior in `bootstrap/app.php`, you can define `report` and `render` methods directly on your exception classes. Laravel will automatically call these methods if they exist:

```
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 as an HTTP response.
     */
    public function render(Request $request): Response
    {
        return response(/* ... */);
    }
}
```

If your exception extends an existing renderable exception (like a Laravel or Symfony exception), you may return `false` from its `render` method to fallback to the default HTTP response:

```
public function render(Request $request): Response|bool
{
    if (/* Determine if custom rendering is needed */) {
        return response(/* ... */);
    }
    return false;
}
```

Similarly, if your exception contains custom reporting logic conditioned on certain conditions, you may return `false` from the `report` method to instruct Laravel to proceed with default reporting:

```
public function report(): bool
{
    if (/* Determine if custom reporting is needed */) {
        // ...
        return true;
    }
    return false;
}
```

Throttling Reported Exceptions

To limit how often exceptions are logged or sent to external tracking, you can use the `throttle` method:

```
use Illuminate\Support\Lottery;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        return Lottery::odds(1, 1000);
    });
});
```

You can conditionally sample exceptions based on their type:

```
use App\Exceptions\ApiMonitoringException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof ApiMonitoringException) {
            return Lottery::odds(1, 1000);
        }
    });
});
```

You may also rate-limit exceptions using the `Limit` class, which is useful to prevent flooding logs, especially during downtime of external services:

```
use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof BroadcastException) {
            return Limit::perMinute(300);
        }
    });
});
```

You can customize the rate limiting key via the `by` method:

```
use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof BroadcastException) {
            return Limit::perMinute(300)->by($e->getMessage());
        }
    });
});
```

You can combine multiple behaviors for different exception types:

```
use App\Exceptions\ApiMonitoringException;
use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Lottery;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        return match (true) {
            $e instanceof BroadcastException => Limit::perMinute(300),
            $e instanceof ApiMonitoringException => Lottery::odds(1, 1000),
            default => Limit::none(),
        };
    });
});
```

HTTP Exceptions

Some exceptions relate to HTTP error codes like 404 (not found), 401 (unauthorized), or 500 (server error). To generate such responses, you can use the `abort` helper:

```
abort(404);
```

Custom HTTP Error Pages

Laravel allows you to display custom error pages by creating view templates in `resources/views/errors/`. For example, to customize the 404 error page, create `404.blade.php`. This view has access to an `$exception` variable:

```
<h2>{{ $exception->getMessage() }}</h2>
```

You can publish the default error page templates via Artisan:

```
php artisan vendor:publish --tag=laravel-errors
```

Fallback HTTP Error Pages

You may define fallback pages like `4xx.blade.php` and `5xx.blade.php` for HTTP status codes that don't have specific views. These fallback pages are rendered when no specific error page exists.

Note: These fallback pages do not affect the responses for 404, 500, and 503 since Laravel has dedicated pages for these codes. For custom pages for these specific errors, create individual templates.

Logging

Introduction

To help you learn more about what's happening within your application, Laravel provides robust logging services that allow you to log messages to files, the system error log, and even to Slack to notify your entire team.

Laravel logging is based on "channels". Each channel represents a specific way of writing log information. For example, the `single` channel writes log files to a single log file, while the `slack` channel sends log messages to Slack. Log messages may be written to multiple channels based on their severity.

Under the hood, Laravel utilizes the Monolog 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 that control 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 building log stacks documentation below.

Available Channel Drivers

Each log channel is powered by a "driver". The driver determines how and where the log message is actually recorded. The following log channel drivers are available in every Laravel application. An entry for most of these drivers is already present in your application's `config/logging.php` configuration file, so be sure to review this file to become familiar with its contents:

Name	Description
`custom`	A driver that calls a specified factory to create a channel.
`daily`	A `RotatingFileHandler` based Monolog driver which rotates daily.
`errorlog`	An `ErrorLogHandler` based Monolog driver.
`monolog`	A Monolog factory driver that may use any supported Monolog handler.
`papertrail`	A `SyslogUdpHandler` based Monolog driver.
`single`	A single file or path based logger channel (`StreamHandler`).
`slack`	A `SlackWebhookHandler` based Monolog driver.
`stack`	A wrapper to facilitate creating "multi-channel" channels.
`syslog`	A `SyslogHandler` based Monolog driver.

Check out the documentation on advanced channel customization to learn more about the `monolog` and `custom` drivers.

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, you may add a `name` option to your channel's configuration:

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

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
`bubble`	Indicates if messages should bubble up to other channels after being handled.	`true`
`locking`	Attempt to lock the log file before writing to it.	`false`
`permission`	The log file's permissions.	`0644`

Additionally, the retention policy for the `daily` channel can be configured via the `LOG_DAILY_DAYS` environment variable or by setting the `days` configuration option:

```
'days' => 14,
```

Configuring the Papertrail Channel

The `papertrail` channel requires `host` and `port` configuration options. These may be defined via the `PAPERTRAIL_URL` and `PAPERTRAIL_PORT` environment variables. You can obtain these values from Papertrail.

Configuring the Slack Channel

The `slack` channel requires a `url` configuration option, which can be set via the `LOG_SLACK_WEBHOOK_URL` environment variable. This URL should be for an incoming webhook you've configured for your Slack team.

By default, Slack will only receive logs at the `critical` level and above; you can adjust this using the `LOG_LEVEL` environment variable or by modifying the `level` configuration option within your Slack log channel configuration.

Logging Deprecation Warnings

PHP, Laravel, and other libraries often notify their users that some features have been deprecated and will be removed in a future version. To log these deprecation warnings, you may specify your preferred `deprecations` log channel via the `LOG_DEPRECATIONS_CHANNEL` environment variable or in `config/logging.php`:

```
'deprecations' => [
    'channel' => env('LOG_DEPRECATIONS_CHANNEL', null),
    'trace' => env('LOG_DEPRECATIONS_TRACE', false),
],
```

Or, you may define a log channel named `deprecations`. If such a channel exists, it will always be used to log deprecations:

```
'channels' => [
    'deprecations' => [
        'driver' => 'single',
        'path' => storage_path('logs/php-deprecation-warnings.log'),
    ],
],
```

Building Log Stacks

The `stack` driver allows you to combine multiple channels into a single log channel. Here's an example configuration that may be used in production:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
        'ignore_exceptions' => false,
    ],
    'syslog' => [
        'driver' => 'syslog',
        'level' => env('LOG_LEVEL', 'debug'),
        'facility' => env('LOG_SYSLOG_FACILITY', LOG_USER),
        'replace_placeholders' => true,
    ],
    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => env('LOG_SLACK_USERNAME', 'Laravel Log'),
        'emoji' => env('LOG_SLACK_EMOJI', ':boom:'),
        'level' => env('LOG_LEVEL', 'critical'),
        'replace_placeholders' => true,
    ],
],
```

This configuration shows how the `stack` channel aggregates `syslog` and `slack`. When logging messages, both channels will be invoked, depending on the message severity and the `level` thresholds.

Log Levels

On each channel, the `level` option determines the minimum severity a message must have to be logged by that channel. Laravel's Monolog supports levels from RFC 5424: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, **debug**.

For example, you can log using:

```
Log::debug('An informational message.');
Log::emergency('The system is down!');
```

- `debug` messages are the least severe and are logged only if the channel's level is `debug` or below.
- `emergency` messages are the most severe and will always be logged if the level allows.

Writing Log Messages

You can write logs using the `Log` facade, which supports multiple levels:

```
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);
```

Example within a controller:

```
<?php

namespace App\Http\Controllers;

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

Additional data can be passed as an array to provide context. This data will be formatted and displayed with the message:

```
use Illuminate\Support\Facades\Log;

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

You can also pass contextual data globally to all subsequent logs using `withContext`:

```
<?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.
     */
    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;
    }
}
```

Or, to share context across all channels:

```
<?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.
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::shareContext([
            'request-id' => $requestId
        ]);

        // ...
    }
}
```

All classes in the `tap` array are resolved by the service container, enabling constructor injection.

Writing to Specific Channels

To log to other channels besides the default, use the `channel` method:

```
use Illuminate\Support\Facades\Log;

Log::channel('slack')->info('Something happened!');
```

You can also create a custom logging stack at runtime:

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

On-Demand Channels

It is possible to create an explicit, one-time channel:

```
use Illuminate\Support\Facades\Log;

Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
])->info('Something happened!');
```

Or, include such a channel in a stack:

```
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

To get fine-grained control over Monolog instances, you can define a `tap` class in your channel's configuration:

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
    'path' => storage_path('logs/laravel.log'),
    'level' => env('LOG_LEVEL', 'debug'),
    'replace_placeholders' => true,
],
```

Create the class with an `__invoke` method that receives an `Illuminate\Log\Logger`:

```
<?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 tap classes are resolved from the container, supporting dependencies via constructor injection.

Creating Monolog Handler Channels

You can define channels that instantiate specific Monolog handlers with custom options:

```
'logentries' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\SyslogUdpHandler::class,
    'handler_with' => [
        'host' => 'my.logentries.internal.datahubhost.company.com',
        'port' => '10000',
    ],
],
```

Monolog Formatters

The default formatter is `LineFormatter`. You can specify a different formatter with `formatter` and `formatter_with`:

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

If a handler supports its own formatter, you can set `'formatter' => 'default'`.

Monolog Processors

You can specify processors that modify messages before logging:

```
'memory' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\StreamHandler::class,
    'handler_with' => [
        'stream' => 'php://stderr',
    ],
    'processors' => [
```

```
        Monolog\Processor\MemoryUsageProcessor::class,
        [
            'processor' => Monolog\Processor\PsrLogMessageProcessor::class,
            'with' => ['removeUsedContextFields' => true],
        ],
    ],
],
```

Creating Custom Channels via Factories

For full control, define a `custom` driver with a `via` option:

```
'example-custom-channel' => [
    'driver' => 'custom',
    'via' => App\Logging\CreateCustomLogger::class,
],
```

Implement the factory class:

```
<?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

Pail is a package to tail logs in real time from the command line, supporting all log drivers.

Installation

Requires PHP 8.2+ and the PCNTL extension:

```
composer require --dev laravel/pail
```

Usage

Run:

```
php artisan pail
```

For verbosity:

```
php artisan pail -v
php artisan pail -vv
```

Press `Ctrl+C` to stop.

Filtering Logs

Use options such as:

- `--filter` : e.g., `php artisan pail --filter="QueryException"`
- `--message` : e.g., `php artisan pail --message="User created"`
- `--level` : e.g., `php artisan pail --level=error`
- `--user` : e.g., `php artisan pail --user=1`

For more options, see the official documentation.

On this page

CSRF Protection

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="user@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. 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 `Illuminate\Foundation\Http\Middleware\ValidateCsrfToken` 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 an 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 Laravel applies to all routes in the `routes/web.php` file. However, you may also exclude specific routes by providing their URIs to the `validateCsrfTokens` method in your application's `bootstrap/app.php` file:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->validateCsrfTokens(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 `Illuminate\Foundation\Http\Middleware\ValidateCsrfToken` middleware, which is included in the `web` middleware group by default, 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

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
```

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)
        ]);
    }
}
```

Defining a route to a controller method

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, it is sometimes convenient to extend a base controller class that contains methods that should be shared across all of your controllers.

Single Action Controllers

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

```
<?php

namespace App\Http\Controllers;

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:

```
use App\Http\Controllers\UserController;

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

Or, you may find it convenient to specify middleware within your controller class. To do so, your controller should implement the `HasMiddleware` interface, which dictates that the controller should have a static `middleware` method. From this method, you may return an array of middleware that should be applied to the controller's actions:

```
<?php

namespace App\Http\Controllers;
```

```
use Illuminate\Routing\Controllers\HasMiddleware;
use Illuminate\Routing\Controllers\Middleware;

class UserController extends Controller implements HasMiddleware
{
    /**
     * Get the middleware that should be assigned to the controller.
     */
    public static function middleware(): array
    {
        return [
            'auth',
            new Middleware('log', only: ['index']),
            new Middleware('subscribed', except: ['store']),
        ];
    }

    // ...
}
```

You may also define controller middleware as closures, which provides a convenient way to define an inline middleware without writing an entire middleware class:

```
<?php

use Closure;
use Illuminate\Http\Request;

/**
 * Get the middleware that should be assigned to the controller.
 */
public static function middleware(): array
{
    return [
        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:

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

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 cannot 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:

```
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. To nest resource controllers, you can use "dot" notation:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);
```

This route will register a nested resource accessible with URIs like:

`/photos/{photo}/comments/{comment}`

Scoping Nested Resources

Laravel's implicit model binding can automatically scope nested bindings so that the resolved child model belongs to the parent. Use the `scoped` method:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

This will register a scoped nested resource accessible via:

`/photos/{photo}/comments/{comment:slug}`

When using a custom keyed implicit binding, Laravel will try to scope the nested query—for example, assuming `comments` is a relationship on `Photo`.

Localizing Resource URIs

By default, `Route::resource` creates URIs with English verbs. To localize `create` and `edit`, use `Route::resourceVerbs()` in your `App\Providers\AppServiceProvider`:

```
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

Laravel's pluralizer supports languages supported for pluralization.

Supplementing Resource Controllers

Add routes outside of the default resource set by defining those routes *before* your `Route::resource()` call to prevent conflicts:

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

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

Singleton Resource Controllers

Some resources are singleton—only one instance exists (e.g., user profile). Register a singleton resource controller:

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

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

This registers routes like:

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

Singleton resources can be nested:

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

Routes will be similar but scoped, e.g.:

`/photos/{photo}/thumbnail`

Additional options, like allowing creation or destruction, are available via methods like `creatable()`, `destroyable()`, etc.

API Singleton Resources

Register via `apiSingleton()`:

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

They can also be made creatable:

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

Middleware and Resource Controllers

Assign middleware globally or per action:

Middleware for all actions

```
use App\Http\Controllers\UserController;

Route::resource('users', UserController::class)
    ->middleware(['auth', 'verified']);
```

```
Route::singleton('profile', ProfileController::class)
    ->middleware('auth');
```

Middleware for specific methods

```
Route::resource('users', UserController::class)
    ->middlewareFor('show', 'auth');

Route::apiResource('users', UserController::class)
    ->middlewareFor(['show', 'update'], 'auth');
```

In combination with singleton controllers:

```
Route::singleton('profile', ProfileController::class)
    ->middlewareFor('show', 'auth');

Route::apiSingleton('profile', ProfileController::class)
    ->middlewareFor(['show', 'update'], 'auth');
```

Excluding middleware on methods

```
Route::middleware(['auth', 'verified', 'subscribed'])->group(function () {
    Route::resource('users', UserController::class)
        ->withoutMiddlewareFor('index', ['auth', 'verified'])
        ->withoutMiddlewareFor(['create', 'store'], 'verified')
        ->withoutMiddlewareFor('destroy', 'subscribed');
});
```

Dependency Injection and Controllers

Constructor Injection

Type-hint dependencies in the constructor; Laravel's container resolves and injects them:

```
<?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

Type-hint dependencies on methods; e.g., the `Request` object:

```php
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        // Store the user...

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

Method injection also supports route parameters:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;

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

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

Hello

Hello

````
# Hello

## Hello

### Hello

#### Hello

##### Hello

###### Hello

Hello

`console.log("Hello")`

[Hello](https://example.com)

![Hello](https://example.com/image.png)

- Hello
- World

1. Hello
2. World

| Hello | World |
| --- | --- |

```console.log("Hello")```

[Hello](https://example.com)
````
````

Hello

Asset Bundling (Vite) - Laravel 12.x - The PHP Framework For Web Artisans

URL Generation

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
```

To generate a URL with query string parameters, you may use the `query` method:

```
echo url()->query('/posts', ['search' => 'Laravel']);
// https://example.com/posts?search=Laravel

echo url()->query('/posts?sort=latest', ['search' => 'Laravel']);
// http://example.com/posts?sort=latest&search=Laravel
```

Providing query string parameters that already exist in the path will overwrite their existing value:

```
echo url()->query('/posts?sort=latest', ['sort' => 'oldest']);
// http://example.com/posts?sort=oldest
```

Arrays of values may also be passed as query parameters. These values will be properly keyed and encoded in the generated URL:

```
$url = url()->query('/posts', ['columns' => ['title', 'body']]);
// http://example.com/posts?columns%5B0%5D=title&columns%5B1%5D=body

echo urldecode($url);
// http://example.com/posts?columns[0]=title&columns[1]=body
```

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();

// Get the path for the previous request...
echo url()->previousPath();
```

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. For example:

```
Route::get('/post/{post}', function (Post $post) {
    // ...
})->name('post.show');
```

To generate a URL to this route, you may use:

```
echo route('post.show', ['post' => 1]);
// http://example.com/post/1
```

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
```

Additional parameters that do not match route parameters will be added as query string parameters:

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);
// http://example.com/post/1?search=rocket
```

Eloquent Models

Often, URLs are generated using an Eloquent model's route key. You can pass models directly, and the helper will extract the route key:

```
echo route('post.show', ['post' => $post]);
```

Signed URLs

Laravel allows you to create signed URLs. These URLs include a "signature" hash to verify the URL has not been tampered with:

```
use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);
```

```
// or without domain
return URL::signedRoute('unsubscribe', ['user' => 1], ['absolute' => false]);
```

To generate a temporary signed route that expires after a specified time:

```
use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

Validating Signed Routes

You can verify the signature in a request:

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (!$request->hasValidSignature()) {
        abort(401);
    }
    // ...
})->name('unsubscribe');
```

You can ignore certain query parameters during validation:

```
if (!$request->hasValidSignatureWhileIgnoring(['page', 'order'])) {
    abort(401);
}
```

Middleware for Signed URLs

Apply middleware to routes for automatic validation:

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed');
```

For URLs without domains:

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed:relative');
```

Handling Invalid Signed Routes

Customize responses for expired or invalid signatures:

```
use Illuminate\Routing\Exceptions\InvalidSignatureException;

->withExceptions(function ($exceptions) {
    $exceptions->render(function (InvalidSignatureException $e) {
        return response()->view('errors.link-expired', [], 403);
    });
});
```

URLs for Controller Actions

The `action` function generates URLs for controller methods:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);

$url = action([UserController::class, 'profile'], ['id' => 1]);
```

Fluent URI Objects

Laravel's `Uri` class provides a fluent interface for creating and manipulating URIs:

```
use Illuminate\Support\Uri;
use App\Http\Controllers\UserController;
use App\Http\Controllers\InvokableController;

// Create from string
$uri = Uri::of('https://example.com/path');

// From paths or routes
$uri = Uri::to('/dashboard');
$uri = Uri::route('users.show', ['user' => 1]);
$uri = Uri::signedRoute('users.show', ['user' => 1]);
$uri = Uri::temporarySignedRoute('user.index', now()->addMinutes(5));
$uri = Uri::action([UserController::class, 'index']);
$uri = Uri::action(InvokableController::class);

// From current request
$uri = $request->uri();
```

Once created, URIs can be modified fluently:

```
$uri = Uri::of('https://example.com')
    ->withScheme('http')
    ->withHost('test.com')
    ->withPort(8000)
    ->withPath('/users')
    ->withQuery(['page' => 2])
    ->withFragment('section-1');
```

For more information, see URI documentation.

Default Values

You can set default URL parameters, such as locale, to be used automatically:

```
// Example of route with locale parameter
Route::get('/{locale}/posts', function () {
    // ...
})->name('post.index');
```

To avoid passing `locale` explicitly each time, you can set defaults:

```
// Middleware to set default URL parameters
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls
{
    public function handle(Request $request, Closure $next)
    {
        URL::defaults(['locale' => $request->user()->locale]);
        return $next($request);
    }
}
```

Middleware Priority

Ensure your middleware runs before Laravel's default `SubstituteBindings`:

```
// In bootstrap/app.php
->withMiddleware(function ($middleware) {
    $middleware->prependToPriorityList(
        before: \Illuminate\Routing\Middleware\SubstituteBindings::class,
        prepend: \App\Http\Middleware\SetDefaultLocaleForUrls::class,
    );
});
```

Hello

Hello

Hello

Hello

Hello

Hello

Package Development

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 Pest 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 to enhance a Laravel application. This guide primarily covers the development of Laravel-specific packages.

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 developing packages, your package will not typically have access to all of Laravel's testing helpers. If you want to test your package as if installed inside a typical Laravel app, you may use the Orchestral Testbench package.

Package Discovery

A Laravel application's `bootstrap/providers.php` contains the list of service providers Laravel loads. Instead of requiring manual addition, you can specify the provider in the `extra` section of your package's `composer.json` so Laravel loads it automatically.

To do this, add to your `composer.json`:

```
"extra": {
    "laravel": {
        "providers": [
            "Barryvdh\\Debugbar\\ServiceProvider"
        ],
        "aliases": {
            "Debugbar": "Barryvdh\\Debugbar\\Facade"
        }
    }
}
```

Once configured, Laravel will automatically register these when your package is installed, streamlining user setup.

Opting Out of Package Discovery

If you want to disable discovery for a specific package, add the package name to the `dont-discover` array in your app's `composer.json`:

```
"extra": {
    "laravel": {
        "dont-discover": [
```

```
            "barryvdh/laravel-debugbar"
        ]
    }
}
```

To disable discovery for all packages, use `"*"`:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
}
```

Service Providers

Service providers act as the connection point between your package and Laravel. They bind resources into Laravel's service container and load views, configs, and language files.

A service provider extends `Illuminate\Support\ServiceProvider` and implements `register` and `boot` methods. The base class is in the `illuminate/support` package—you should include it as a dependency. For more details, see their documentation.

Resources

Configuration

Publish your package's config file to Laravel's `config` directory with `$this->publishes()` in your provider's `boot` method:

```
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../config/courier.php' => config_path('courier.php'),
    ]);
}
```

Users can then run `php artisan vendor:publish` to copy config file for customization. Access config values normally via `config('courier.option')`.

Note: Avoid defining closures in config files, as they cannot be serialized when caching.

Default Package Configuration

Use `mergeConfigFrom()` in your provider's `register` method to merge default config:

```
public function register(): void
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```

This merges only the first level of the array. Partial multilevel arrays require manual merging for nested options.

Routes

If your package has routes, load them with `$this->loadRoutesFrom()` in `boot()`:

```
public function boot(): void
{
    $this->loadRoutesFrom(__DIR__.'/../routes/web.php');
}
```

Migrations

Publish migrations with `$this->publishesMigrations()` in `boot()`:

```
public function boot(): void
{
    $this->publishesMigrations([
        __DIR__.'/../database/migrations' => database_path('migrations'),
    ]);
}
```

Laravel will update timestamps upon publish.

Language Files

To load translations, in `boot()`:

```
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');
}
```

Package translation lines are accessed via `package::file.line`, e.g., `trans('courier::messages.welcome')`.

For JSON translations:

```
public function boot(): void
{
    $this->loadJsonTranslationsFrom(__DIR__.'/../lang');
}
```

Publish language files to `lang/vendor/courier`:

```
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');

    $this->publishes([
        __DIR__.'/../lang' => $this->app->langPath('vendor/courier'),
    ]);
}
```

Views

Register views with `$this->loadViewsFrom()`:

```
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

Use `package::view` syntax; e.g., `view('courier::dashboard')`.

To override views, place customized ones in `resources/views/vendor/courier/`. Laravel prefers vendor views in this directory when publishing.

Publish views:

```
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

    $this->publishes([
        __DIR__.'/../resources/views' => resource_path('views/vendor/courier'),
    ]);
}
```

View Components

Register Blade components in `boot()`:

```
public function boot(): void
{
    Blade::component('package-alert', AlertComponent::class);
}
```

Use `<x-package-alert />` in views.

Autoloading Components

Use `componentNamespace()` for auto-loaded components in a namespace:

```
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Using `<x-nightshade::calendar />` or `<x-nightshade::color-picker />`.

Anonymous Components

Place anonymous components in `components` directory within your views folder; reference via `package::view`.

"About" Artisan Command

Laravel's `about` command displays environment info. You can add package info via `AboutCommand`.

In your provider's `boot()`:

```
use Illuminate\Foundation\Console\AboutCommand;

public function boot(): void
{
    AboutCommand::add('My Package', fn () => ['Version' => '1.0.0']);
}
```

Commands

Register Artisan commands with `$this->commands()` in `boot()`:

```
public function boot(): void
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            InstallCommand::class,
            NetworkCommand::class,
        ]);
    }
}
```

Optimize Commands

Register commands to run with `php artisan optimize`:

```
public function boot(): void
{
    if ($this->app->runningInConsole()) {
        $this->optimizes(
            optimize: 'package:optimize',
            clear: 'package:clear-optimizations'
        );
    }
}
```

Public Assets

Publish assets like JS, CSS, images with `$this->publishes()`:

```
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../public' => public_path('vendor/courier'),
    ], 'public');
}
```

Publish with `--tag=public --force` to overwrite existing:

```
php artisan vendor:publish --tag=public --force
```

Publishing File Groups

Publish assets/resources separately using tags, e.g., to publish config and migrations:

```
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');
}
```

Publish by tags:

```
php artisan vendor:publish --tag=courier-config
```

Publish all associated files from provider with:

```
php artisan vendor:publish --provider="Your\Package\ServiceProvider"
```

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 it to finish executing, while `start` is used for asynchronous execution. We'll examine both approaches.

Basic Synchronous Process

First, to run a basic, synchronous process and inspect its result:

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

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

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

The `Process::run()` method returns an instance of `Illuminate\Contracts\Process\ProcessResult`, which provides helpful methods:

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

$result->successful();
$result->failed();
$result->exitCode();
$result->output();
$result->errorOutput();
```

Throwing Exceptions on Failure

You can have the process throw an exception if it fails by using `throw` or `throwIf`:

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

$result = Process::run('ls -la')->throwIf($condition);
```

Process Options

You can customize process behavior before invoking it using various methods:

Working Directory Path

Specify the working directory:

```
$result = Process::path(__DIR__)->run('ls -la');
```

Input

Provide input via standard input:

```
$result = Process::input('Hello World')->run('cat');
```

Timeouts

Set a timeout (default is 60 seconds):

```
$result = Process::timeout(120)->run('bash import.sh');
```

To disable timeout:

```
$result = Process::forever()->run('bash import.sh');
```

Set idle timeout (max seconds without output):

```
$result = Process::timeout(60)->idleTimeout(30)->run('bash import.sh');
```

Environment Variables

Provide environment variables:

```
$result = Process::forever()
    ->env(['IMPORT_PATH' => __DIR__])
    ->run('bash import.sh');
```

To remove an inherited variable, set it to `false`:

```
$result = Process::forever()
    ->env(['LOAD_PATH' => false])
    ->run('bash import.sh');
```

TTY Mode

Enable TTY mode (connect process input/output directly):

```
Process::forever()->tty()->run('vim');
```

Process Output

Access output via `output()` and `errorOutput()`:

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

echo $result->output();
echo $result->errorOutput();
```

Real-Time Output with Callback

Gather output in real-time with a closure:

```
$results = Process::run('ls -la', function (string $type, string $output) {
    echo $output;
});
```

Output Matching

Check if output contains a string:

```
if (Process::run('ls -la')->seeInOutput('laravel')) {
    // ...
}
```

Disable Output Gathering

Prevent memory use by disabling output:

```
$result = Process::quietly()->run('bash import.sh');
```

Pipelines

Make the output of one process the input of another:

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

Or, pass command strings directly:

```
$result = Process::pipe([
    'cat example.txt',
    'grep -i "laravel"',
]);
```

Real-Time Output in Pipelines

Use a closure:

```
$result = Process::pipe(function (Pipe $pipe) {
    $pipe->command('cat example.txt');
    $pipe->command('grep -i "laravel"');
}, function (string $type, string $output) {
    echo $output;
});
```

Named Processes in Pipeline

Assign string keys to processes:

```
$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) {
    // ...
});
```

Access results:

```
$results = $pipe->wait();

return $results['first']->output();
```

Pool Processes

Manage multiple concurrent asynchronous processes:

```
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();
```

Access individual process IDs:

```
$processIds = $pool->running()->each->id();
```

Send signals to all pool processes:

```
$pool->signal(SIGUSR2);
```

Testing

Laravel's process testing features include:

Faking Processes

Faking all processes:

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

Process::fake();

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

```
Process::assertRan('bash import.sh');

Process::assertRan(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'bash import.sh' && $process->timeout === 60;
});
```

Faking Specific Processes

Set fake results for particular commands:

```
Process::fake([
    'cat *' => Process::result(
        output: 'Test "cat" output',
    ),
    'ls *' => Process::result(
        output: 'Test "ls" output',
    ),
]);
```

Or, specify simple strings:

```
Process::fake([
    'cat *' => 'Test "cat" output',
    'ls *' => 'Test "ls" output',
]);
```

Fake Multiple Invocations

Use `sequence()` to assign different fakes for repeated calls:

```
Process::fake([
    'ls *' => Process::sequence()
        ->push(Process::result('First invocation'))
        ->push(Process::result('Second invocation')),
]);
```

Fake Asynchronous Lifecycles

Describe how a fake process behaves over multiple iterations:

```
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),
]);
```

Assertions

Check if process was invoked:

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

Process::assertRan(function (PendingProcess $process, ProcessResult $result) {
```

```
    return $process->command === 'ls -la' && $process->path === __DIR__ && $process->t
});
```

Check process was not invoked:

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

Process::assertDidntRun(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'ls -la';
});
```

Check invocation count:

```
Process::assertRanTimes('ls -la', times: 3);

Process::assertRanTimes(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'ls -la';
}, times: 3);
```

Prevent Stray Processes

Avoid actual process startups in tests:

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

Process::preventStrayProcesses();

Process::fake([
    'ls *' => 'Test output...',
]);

Process::run('ls -la'); // returns fake result
Process::run('bash import.sh'); // throws exception
```

Hello

Rate Limiting

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:

```
// Example configuration
'default' => env('CACHE_STORE', 'database'),

'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 rate limits a given callback for a given number of seconds.

The `attempt` method returns `false` when the callback has no remaining attempts available; otherwise, it returns the callback's result or `true`. The first argument accepted by `attempt` is a rate limiter "key", which can be any string representing 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 `attempt`, which is the "decay rate" — the number of seconds until attempts are reset. For example, to allow five attempts every two minutes:

```
$executed = RateLimiter::attempt(
    'send-message:' . $user->id,
    $perTwoMinutes = 5,
    function() {
        // Send message...
```

```
    },
    $decayRate = 120,
);
```

Manually Incrementing Attempts

You can manually interact with the rate limiter using methods like `tooManyAttempts`, `increment`, and `remaining`.

Check if Limits are Exceeded

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:' . $user->id, $perMinute = 5)) {
    return 'Too many attempts!';
}
```

Increment Attempts

```
RateLimiter::increment('send-message:' . $user->id);

// Proceed to send message...
```

Remaining Attempts

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::remaining('send-message:' . $user->id, $perMinute = 5)) {
    RateLimiter::increment('send-message:' . $user->id);

    // Send message...
}
```

Increment by a Specific Amount

To increment attempts by more than one, pass the amount as a second parameter:

```
RateLimiter::increment('send-message:' . $user->id, amount: 5);
```

Determining Limiter Availability

When no attempts are left, `availableIn` returns the seconds until more attempts are 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);

// Proceed to send message...
```

Clearing Attempts

You can reset the attempt count for a specific key using `clear`. For example, when a message is read:

```
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

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

- `__()`
  The `__` function translates the given translation string or translation key using your localization files. Example:

  ```
  echo __('Welcome to our application');
  echo __('messages.welcome');
  ```

  If the translation does not exist, it returns the key itself.

- `class_basename()`
  Returns the class name of the given class without its namespace.

- `e()`
  Runs PHP's `htmlspecialchars` with double encoding.

- `preg_replace_array()`
  Replaces sequentially in a string using an array of replacements.

- `Str::after`
  Gets the string after a given substring.

- `Str::afterLast`
  Gets everything after the last occurrence of a substring.

- `Str::apa`
  Converts a string to APA title case.

- `Str::ascii`
  Transliterates a string into ASCII.

- `Str::before`
  Gets everything before a substring.

- `Str::beforeLast`
  Gets everything before the last occurrence of a substring.

- `Str::between`
  Extracts the string between two substrings.

- `Str::betweenFirst`
  Extracts the shortest substring between two strings.

- `Str::camel`
  Converts string to camelCase.
- `Str::charAt`
  Gets a character at a specific position (returns false if out of bounds).
- `Str::chopStart`
  Removes specified start substring if present.
- `Str::chopEnd`
  Removes specified end substring if present.
- `Str::contains`
  Checks if a string contains a value or any of multiple values.
- `Str::containsAll`
  Checks if a string contains all given substrings.
- `Str::doesntContain`
  Checks if a string does not contain a value or any of multiple values.
- `Str::doesntEndWith`
  Checks if a string does not end with a value or any in multiple.
- `Str::doesntStartWith`
  Checks if a string does not start with a value or any in multiple.
- `Str::excerpt`
  Extracts an excerpt matching the first occurrence of a phrase.
- `Str::finish`
  Appends a suffix if not already present.
- `Str::fromBase64`
  Decodes a Base64 string.
- `Str::headline`
  Converts delimited strings into headline format with capitalized words.
- `Str::inlineMarkdown`
  Converts Markdown to inline HTML (without wrapping in block elements).
- `Str::is`
  Pattern-matching with wildcards.
- `Str::isAscii`
  Checks if a string is ASCII.
- `Str::isJson`
  Checks if a string contains valid JSON.
- `Str::isUlid`
  Checks if a string is a ULID.
- `Str::isUrl`
  Checks if a string is a URL; can specify protocols.
- `Str::isUuid`
  Checks if a string is a UUID.

- `Str::kebab`
  Converts string to kebab-case.
- `Str::lcfirst`
  Lowercases the first character.
- `Str::length`
  Gets string length.
- `Str::limit`
  Truncates string to specified length, with options for omission and preserving words.
- `Str::lower`
  Converts string to lowercase.
- `Str::markdown`
  Converts Markdown to HTML.
- `Str::mask`
  Masks part of a string, obfuscating data like emails and phone numbers.
- `Str::match`
  Extracts part of a string matching regex.
- `Str::matchAll`
  Returns all matches of a regex pattern.
- `Str::orderedUuid`
  Generates timestamp-ordered UUID.
- `Str::padBoth`
  Wraps string on both sides with padding.
- `Str::padLeft`
  Pads string on the left.
- `Str::padRight`
  Pads string on the right.
- `Str::password`
  Generates a secure random password.
- `Str::plural`
  Converts to plural form, supporting multiple languages.
- `Str::pluralStudly`
  Converts singular studly-case words to plural.
- `Str::position`
  Finds the position of a substring.
- `Str::random`
  Generates a random string.
- `Str::remove`
  Removes specified substring(s).
- `Str::repeat`
  Repeats the string.

- `Str::replace`
  Replaces a string within another.
- `Str::replaceArray`
  Replaces sequentially using an array.
- `Str::replaceFirst`
  Replaces first occurrence.
- `Str::replaceLast`
  Replaces last occurrence.
- `Str::replaceMatches`
  Replaces regex-matched parts.
- `Str::replaceStart`
  Replaces start substring if present.
- `Str::replaceEnd`
  Replaces end substring if present.
- `Str::reverse`
  Reverses the string.
- `Str::singular`
  Converts to singular form.
- `Str::slug`
  Creates URL slug.
- `Str::snake`
  Converts to snake_case.
- `Str::split`
  Splits into a collection with regex.
- `Str::squish`
  Removes excess whitespace.
- `Str::start`
  Prepends string if not start with specified value.
- `Str::startsWith`
  Checks if string starts with a value or any in array.
- `Str::studly`
  Converts to StudlyCase.
- `Str::substr`
  Extracts part by start and length.
- `Str::substrReplace`
  Replaces substring at position.
- `Str::swap`
  Replaces multiple values according to array in string.
- `Str::take`
  Gets first characters.

- `Str::tap`
  Allows examining string without modifying.
- `Str::test`
  Tests pattern match.
- `Str::title`
  Converts to Title Case.
- `Str::toBase64`
  Encodes string as Base64.
- `Str::toHtmlString`
  Converts to `HtmlString`.
- `Str::toUri`
  Converts to Uri object.
- `Str::transliterate`
  Converts to ASCII approximation.
- `Str::trim`
  Trims whitespace or characters.
- `Str::ltrim`
  Trims left side.
- `Str::rtrim`
  Trims right side.
- `Str::ucfirst`
  Uppercases first character.
- `Str::ucsplit`
  Splits into array at uppercase letters.
- `Str::unwrap`
  Removes specified start/end strings.
- `Str::upper`
  Uppercases string.
- `Str::when`
  Executes closure if condition is true.
- `Str::whenContains`
  Executes if string contains value.
- `Str::whenContainsAll`
  Executes if string contains all values.
- `Str::whenDoesntEndWith`
  Executes if not ending with substring.
- `Str::whenDoesntStartWith`
  Executes if not starting with substring.
- `Str::whenEmpty`
  Executes if empty.

- `Str::whenNotEmpty`
  Executes if not empty.
- `Str::whenStartsWith`
  Executes if start with.
- `Str::whenEndsWith`
  Executes if ending with.
- `Str::whenExactly`
  Executes if exactly matches.
- `Str::whenNotExactly`
  Executes if not exactly matching.
- `Str::whenIs`
  Executes if pattern matches.
- `Str::whenIsAscii`
  Executes if string is ASCII.
- `Str::whenIsUlid`
  Executes if string is ULID.
- `Str::whenIsUuid`
  Executes if string is UUID.
- `Str::whenTest`
  Executes if regex pattern matches.
- `Str::wordCount`
  Counts words in a string.
- `Str::words`
  Limits to number of words with optional suffix.
- `Str::wrap`
  Wraps string with prefix and/or suffix.

On this page

Hello

Cache

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 store 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. Additionally, 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 other options that you may review. By default, Laravel is configured to use the `database` cache driver, which stores serialized, cached objects in your application's database.

Driver Prerequisites

Database

When using the `database` cache driver, you will need a database table to contain the cache data. Typically, this is included in Laravel's default `0001_01_01_000001_create_cache_table.php` migration; however, if your application does not contain this migration, you may use the Artisan command to create it:

```
php artisan make:cache-table
php artisan migrate
```

Memcached

Using the Memcached driver requires the Memcached PECL package to be installed. You may list all of your Memcached servers in `config/cache.php`. The file already contains an 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 can set the `host` option to a UNIX socket path. In this case, set the `port` to `0`:

```
'memcached' => [
    // ...
```

```
    'servers' => [
        [
            'host' => '/var/run/memcached/memcached.sock',
            'port' => 0,
            'weight' => 100,
        ],
    ],
],
```

Redis

Before using Redis, install the PhpRedis PHP extension via PECL or the `predis/predis` package (~2.0) via Composer. Laravel Sail includes this extension. Official Laravel hosting platforms such as Laravel Cloud and Laravel Forge have PhpRedis installed by default.

For more information, see the Laravel Redis configuration.

DynamoDB

Before using the DynamoDB cache driver, you need to create a DynamoDB table to store all cached data, typically named `cache`. The table name can be set via the `stores.dynamodb.table` configuration option or the `DYNAMODB_CACHE_TABLE` environment variable.

The DynamoDB table should have a string partition key named `key` by default or as specified in the configuration. You should also enable TTL (Time to Live) on the table, setting the TTL attribute to `expires_at`. Install the AWS SDK using:

```
composer require aws/aws-sdk-php
```

Configure DynamoDB in your application's `cache.php`:

```
'dynamodb' => [
    'driver' => 'dynamodb',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => env('DYNAMODB_CACHE_TABLE', 'cache'),
    'endpoint' => env('DYNAMODB_ENDPOINT'),
],
```

MongoDB

If using MongoDB, a `mongodb` cache driver is provided by the `mongodb/laravel-mongodb` package. MongoDB supports TTL indexes to automatically clear expired cache items. For configuration, see the MongoDB cache documentation.

Cache Usage

Obtaining a Cache Instance

Use the `Cache` facade, which provides convenient access to cache implementations:

```
<?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

Use the `store` method to select specific cache stores:

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

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

Retrieving Items From the Cache

Use `Cache::get()`, which returns `null` if the item doesn't exist. Provide a default value as second argument if desired:

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

You can pass a closure as the default value; the closure's result will be returned if the item isn't in cache:

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

Determining Item Existence

Use `has()` to check if an item exists (returns `false` if the item exists but its value is `null`):

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

Incrementing / Decrementing Values

Adjust integer cache items:

```
// 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 (`remember` pattern)

Retrieve an item or execute a callback to store if missing:

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

Use `rememberForever()` to store indefinitely:

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

Stale While Revalidate

Serve partially stale data while background recalculation occurs:

```
$value = Cache::flexible('users', [5, 10], function () {
    return DB::table('users')->get();
});
```

Retrieve and Delete (`pull`)

Retrieve and remove an item:

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

Storing Items

Use `put()`:

```
Cache::put('key', 'value', $seconds);
Cache::put('key', 'value'); // forever if no seconds specified
Cache::put('key', 'value', now()->addMinutes(10));
```

Store if Not Present (`add()`)

Adds only if the key does not exist, returns `true` if successful:

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

Store Forever (`forever()`)

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

> If using Memcached, forever items may be removed when cache reaches size limit.

Removing Items

Use `forget()`:

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

Or set to expire immediately with non-positive seconds:

```
Cache::put('key', 'value', 0);
Cache::put('key', 'value', -5);
```

Clear Entire Cache

```
Cache::flush();
```

> Flushing does not respect cache prefixes; it removes all entries.

Cache Memoization (`memo` Driver)

Stores cache values in memory during request/job:

```
use Illuminate\Support\Facades\Cache;

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

You can specify a cache store:

```
// Default store
$value = Cache::memo()->get('key');
// Redis store
$value = Cache::memo('redis')->get('key');
```

Repeated calls within the same request will use in-memory cached value, avoiding hits to the underlying store.

Mutating cache via `put()`, `increment()`, etc., automatically forgets in-memory cache:

```
Cache::memo()->put('name', 'Taylor');
Cache::memo()->get('name'); // hits underlying cache...

Cache::memo()->put('name', 'Tim'); // forgets in-memory cache and updates underlying c
```

The Cache Helper

Use the global `cache()` function:

```
$value = cache('key'); // get

cache(['key' => 'value'], $seconds); // store

cache(['key' => 'value'], now()->addMinutes(10));
```

Calling `cache()` with no arguments returns a cache repository instance, allowing other cache methods:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

You can mock cache in tests with `Cache::shouldReceive()`.

Atomic Locks

These are distributed locks to prevent race conditions, used with cache drivers like Memcached, Redis, DynamoDB, database, file, or array.

Managing Locks

Create or manage locks via `Cache::lock()`:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...
    $lock->release();
}
```

`get()` also accepts a closure; Laravel releases lock after closure executes:

```
Cache::lock('foo', 10)->get(function () {
    // Lock acquired for 10 seconds and released after...
});
```

Wait for Lock

To wait for a lock with timeout:

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5); // max 5 seconds
    // Lock acquired after waiting...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    $lock->release();
}
```

Or with a closure and automatic release:

```
Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired for 10s after waiting up to 5s...
});
```

Managing Locks Across Processes

You may acquire a lock in one process and release it in another by passing the lock's owner token:

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

In the job:

```
Cache::restoreLock('processing', $this->owner)->release();
```

To force release regardless of owner:

```
Cache::lock('processing')->forceRelease();
```

Adding Custom Cache Drivers

Writing the Driver

Implement the `Illuminate\Contracts\Cache\Store` contract. Example for MongoDB:

```
<?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() {}
}
```

Registering the driver:

```
Cache::extend('mongo', function (Application $app) {
    return Cache::repository(new MongoStore);
});
```

Place your custom cache code in a suitable namespace, e.g., `app/Extensions`.

Registering the Driver

Register your custom driver within a `booting` callback in a service provider:

```
<?php
namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Contracts\Foundation\Applicaton;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->booting(function () {
            Cache::extend('mongo', function (Application $app) {
                return Cache::repository(new MongoStore);
            });
```

```
        });
    }
    public function boot(): void
    {
        // ...
    }
}
```

Update the environment variable or config option `CACHE_STORE` to your new driver name.

Events

To run code on cache events, listen for Laravel's cache events such as:

- `Illuminate\Cache\Events\CacheFlushed`
- `Illuminate\Cache\Events\CacheHit`
- `Illuminate\Cache\Events\CacheMissed`
- etc.

Disable cache events by setting `events` to `false` in `config/cache.php`, e.g.,

```
'database' => [
    'driver' => 'database',
    // ...
    'events' => false,
],
```

Hello

Concurrency

Introduction

Sometimes you may need to execute several slow tasks which do not depend on one another. In many cases, significant performance improvements can be realized by executing the tasks concurrently. Laravel's `Concurrency` facade provides a simple, convenient API for executing closures concurrently.

How it Works

Laravel achieves concurrency by serializing the given closures and dispatching them to a hidden Artisan CLI command, which unserializes the closures and invokes it within its own PHP process. After the closure has been invoked, the resulting value is serialized back to the parent process.

The `Concurrency` facade supports three drivers: `process` (the default), `fork`, and `sync`.

The `fork` driver offers improved performance compared to the default `process` driver, but it may only be used within PHP's CLI context, as PHP does not support forking during web requests. Before using the `fork` driver, you need to install the `spatie/fork` package:

```
composer require spatie/fork
```

The `sync` driver is primarily useful during testing when you want to disable all concurrency and simply execute the given closures in sequence within the parent process.

Running Concurrent Tasks

To run concurrent tasks, you may invoke the `Concurrency` facade's `run` method. The `run` method accepts an array of closures which should be executed simultaneously in child PHP processes:

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

[$userCount, $orderCount] = Concurrency::run([
    fn () => DB::table('users')->count(),
    fn () => DB::table('orders')->count(),
]);
```

To use a specific driver, you may use the `driver` method:

```
$results = Concurrency::driver('fork')->run(...);
```

Or, to change the default concurrency driver, you should publish the `concurrency` configuration file via the `config:publish` Artisan command and update the `default` option within the file:

```
php artisan config:publish concurrency
```

Deferring Concurrent Tasks

If you would like to execute an array of closures concurrently, but are not interested in the results returned by those closures, you should consider using the `defer` method. When the `defer` method is invoked, the given closures are not executed immediately. Instead, Laravel will execute the closures concurrently after the HTTP response has been sent to the user:

```
use App\Services\Metrics;
use Illuminate\Support\Facades\Concurrency;

Concurrency::defer([
    fn () => Metrics::report('users'),
    fn () => Metrics::report('orders'),
]);
```

On this page

Hello

Contracts

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 include them in your class's 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 others 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. They are not mutually exclusive; some parts of your application may use facades while others depend on contracts. As long as you keep your class responsibilities focused, you'll notice very few practical differences.

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 requiring Laravel's concrete implementations in your package's `composer.json`.

How to Use Contracts

Many types of classes in Laravel are resolved through the service container, including controllers, event listeners, middleware, queued jobs, and route closures. To get an implementation of a contract, you can just *type-hint* the interface in the constructor of the class being resolved.

Example: Event Listener

```
<?php

namespace App\Listeners;
```

```
use App\Events\OrderWasPlaced;
use App\Models\User;
use Illuminate\Contracts\Redis\Factory;

class CacheOrderInformation
{
    /**
     * Create the event listener.
     */
    public function __construct(
        protected Factory $redis,
    ) {}

    /**
     * Handle the event.
     */
    public function handle(OrderWasPlaced $event): void
    {
        // ...
    }
}
```

When the event listener is resolved, the service container reads the type-hints on the constructor and injects 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\Broadcasting\Broadcaster	`Broadcast::connection()`
Illuminate\Contracts\Broadcasting\Factory	`Broadcast`
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Bus\Dispatcher	`Bus`
Illuminate\Contracts\Bus\QueueingDispatcher	`Bus::dispatchToQueue()`
Illuminate\Contracts\Cache\Factory	`Cache`
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	`Cache::driver()`

Contract	References Facade
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\Mailable	
Illuminate\Contracts\Mail\Mailer	`Mail`
Illuminate\Contracts\Mail\MailQueue	`Mail::queue()`
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	
Illuminate\Contracts\Support\Htmlable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\MessageBag	
Illuminate\Contracts\Support\MessageProvider	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Support\Responsable	

Contract	References Facade
Illuminate\Contracts\Translation\Loader	
Illuminate\Contracts\Translation\Translator	`Lang`
Illuminate\Contracts\Validation\Factory	`Validator`
Illuminate\Contracts\Validation\ValidatesWhenResolved	
Illuminate\Contracts\Validation\ValidationRule	
Illuminate\Contracts\Validation\Validator	`Validator::make()`
Illuminate\Contracts\View\Engine	
Illuminate\Contracts\View\Factory	`View`
Illuminate\Contracts\View\View	`View::make()`

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

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...

Configuration

Laravel's filesystem configuration file is located at `config/filesystems.php`. Within this file...

The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory...

```
use Illuminate\Support\Facades\Storage;

Storage::disk('local')->put('example.txt', 'Contents');
```

The Public Disk

The `public` disk is intended for files accessible publicly...

To create a symbolic link from `storage/app/public` to `public/storage`, run:

```
php artisan storage:link
```

To generate a URL:

```
echo asset('storage/file.txt');
```

Configure additional symbolic links in the `filesystems.php`:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

To remove symbolic links:

```
php artisan storage:unlink
```

Driver Prerequisites

S3 Driver Configuration

Require the Flysystem AWS S3 package:

```
composer require league/flysystem-aws-s3-v3 "^3.0" --with-all-dependencies
```

Configure credentials via environment variables:

```
AWS_ACCESS_KEY_ID=<your-key-id>
AWS_SECRET_ACCESS_KEY=<your-secret-key>
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=<your-bucket-name>
AWS_USE_PATH_STYLE_ENDPOINT=false
```

FTP Driver Configuration

Require the Flysystem FTP package:

```
composer require league/flysystem-ftp "^3.0"
```

Example configuration:

```
'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

Require the Flysystem SFTP package:

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Example configuration:

```
'sftp' => [
  'driver' => 'sftp',
  'host' => env('SFTP_HOST'),

  // Basic auth
  'username' => env('SFTP_USERNAME'),
  'password' => env('SFTP_PASSWORD'),

  // SSH key-based authentication
  'privateKey' => env('SFTP_PRIVATE_KEY'),
  'passphrase' => env('SFTP_PASSPHRASE'),

  // Permissions
  'visibility' => 'private',
  'directory_visibility' => 'private',

  // Optional settings...
  // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
  // 'maxTries' => 4,
  // 'port' => env('SFTP_PORT', 22),
  // 'root' => env('SFTP_ROOT', ''),
  // 'timeout' => 30,
],
```

Scoped and Read-Only Filesystems

Scoped disks prefix all paths. Requires installing:

```
composer require league/flysystem-path-prefixing "^3.0"
```

Example scope:

```
's3-videos' => [
  'driver' => 'scoped',
  'disk' => 's3',
  'prefix' => 'path/to/videos',
],
```

Read-only disks:

```
composer require league/flysystem-read-only "^3.0"
```

And in configuration:

```
's3-videos' => [
  'driver' => 's3',
  // ...
  'read-only' => true,
],
```

Amazon S3 Compatible Filesystems

Default `s3` disk can interact with compatible services like MinIO, DigitalOcean Spaces, etc. Usually, update credentials and set `endpoint`:

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

MinIO

For proper URL generation with MinIO, set:

```
AWS_URL=http://localhost:9000/local
```

Note: URL signed URLs may not work if endpoint isn't accessible from the client.

Obtaining Disk Instances

Use the `Storage` facade:

```
Storage::put('avatars/1', $content);

Storage::disk('s3')->put('avatars/1', $content);
```

On-Demand Disks

Create disks at runtime:

```
$disk = Storage::build([
  'driver' => 'local',
  'root' => '/path/to/root',
]);

$disk->put('image.jpg', $content);
```

Retrieving Files

Use `get`:

```
$contents = Storage::get('file.jpg');
```

Decode JSON file:

```
$orders = Storage::json('orders.json');
```

Check existence:

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

Check missing:

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

Downloading Files

```
return Storage::download('file.jpg');
```

```
return Storage::download('file.jpg', $name, $headers);
```

File URLs

```
$url = Storage::url('file.jpg');
```

Note: For local, URLs are relative (`/storage/file.jpg`). For S3, absolute URL.

Configure URL host:

```
'public' => [
  'driver' => 'local',
  'root' => storage_path('app/public'),
  'url' => env('APP_URL') . '/storage',
  'visibility' => 'public',
  'throw' => false,
],
```

Temporary URLs

Create expiring URLs:

```
$url = Storage::temporaryUrl('file.jpg', now()->addMinutes(5));
```

Enable local temporary URLs (`serve` option):

```
'local' => [
  'driver' => 'local',
  'root' => storage_path('app/private'),
  'serve' => true,
  'throw' => false,
],
```

Specify request parameters:

```
$url = Storage::temporaryUrl(
  'file.jpg',
  now()->addMinutes(5),
  [
    'ResponseContentType' => 'application/octet-stream',
    'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
  ]
);
```

Customize temporary URL generation via `buildTemporaryUrlsUsing` in a service provider:

```
Storage::disk('local')->buildTemporaryUrlsUsing(function ($path, $expiration, $options
    return URL::temporarySignedRoute(
        'files.download',
        $expiration,
        array_merge($options, ['path' => $path])
    );
});
```

Temporary Upload URLs

Supported only via `s3` . Generate with:

```
[$url, $headers] = Storage::temporaryUploadUrl('file.jpg', now()->addMinutes(5));
```

File Metadata

Size:

```
$size = Storage::size('file.jpg');
```

Last modified:

```
$time = Storage::lastModified('file.jpg');
```

MIME type:

```
$mime = Storage::mimeType('file.jpg');
```

File Paths

```
$path = Storage::path('file.jpg');
```

Storing Files

Use `put`:

```
Storage::put('file.jpg', $contents);
```

Store resource:

```
Storage::put('file.jpg', $resource);
```

Handling failed write:

```
if (!Storage::put('file.jpg', $contents)) {
    // handle failure
}
```

Set `throw` to true in disk config to throw exceptions on failures.

Prepending and Appending

```
Storage::prepend('file.log', 'Prepended Text');
Storage::append('file.log', 'Appended Text');
```

Copying and Moving Files

```
Storage::copy('old/file.jpg', 'new/file.jpg');
Storage::move('old/file.jpg', 'new/file.jpg');
```

Automatic Streaming

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

$path = Storage::putFile('photos', new File('/path/to/photo'));
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

You may set visibility during storage:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

File Uploads

Store uploaded files:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserAvatarController extends Controller
{
    public function update(Request $request): string
    {
        $path = $request->file('avatar')->store('avatars');
        return $path;
    }
}
```

Or:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying a File Name

Use `storeAs`:

```
$path = $request->file('avatar')->storeAs('avatars', $request->user()->id);
```

Or:

```
$path = Storage::putFileAs('avatars', $request->file('avatar'), $request->user()->id);
```

Other Uploaded File Info

Original name:

```
$file = $request->file('avatar');
$name = $file->getClientOriginalName();
$extension = $file->getClientOriginalExtension();
```

Hash name for security:

```
$name = $file->hashName();
$extension = $file->extension();
```

File Visibility

Set visibility during store:

```
Storage::put('file.jpg', $contents, 'public');
```

Get/set visibility:

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

Store publicly:

```
$path = $request->file('avatar')->storePublicly('avatars');
$path = $request->file('avatar')->storePubliclyAs('avatars', $request->user()->id);
```

Local files with `public` visibility:

```
// In configuration
'local' => [
  'driver' => 'local',
  'root' => storage_path('app'),
  'permissions' => [
    'file' => [
      'public' => 0644,
      'private' => 0600,
    ],
    'dir' => [
      'public' => 0755,
      'private' => 0700,
    ],
  ],
  'throw' => false,
],
```

Deleting Files

```
Storage::delete('file.jpg');
Storage::delete(['file.jpg', 'file2.jpg']);

Storage::disk('s3')->delete('path/file.jpg');
```

Directories

Get All Files:

```
$files = Storage::files($directory);
$files = Storage::allFiles($directory);
```

Get All Directories:

```
$directories = Storage::directories($directory);
$directories = Storage::allDirectories($directory);
```

Create Directory:

```
Storage::makeDirectory($directory);
```

Delete Directory:

```
Storage::deleteDirectory($directory);
```

Testing

Using `Storage::fake('disk_name')` for tests:

```
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
```

```
test('albums can be uploaded', function () {
    Storage::fake('photos');

    $response = $this->json('POST', '/photos', [
        UploadedFile::fake()->image('photo1.jpg'),
        UploadedFile::fake()->image('photo2.jpg')
    ]);

    Storage::disk('photos')->assertExists('photo1.jpg');
    Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

    Storage::disk('photos')->assertMissing('missing.jpg');
    Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);

    Storage::disk('photos')->assertCount('/wallpapers', 2);
    Storage::disk('photos')->assertDirectoryEmpty('/wallpapers');
});
```

Custom Filesystems

Add a custom driver, e.g., Dropbox:

```
composer require spatie/flysystem-dropbox
```

Register in a service provider:

```
namespace App\Providers;

use Illuminate\Contracts\Foundation\Applicaton;
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
{
    public function register(): void
    {
        // ...
    }

    public function boot(): void
    {
        Storage::extend('dropbox', function (Application $app, array $config) {
            $adapter = new DropboxAdapter(new DropboxClient($config['authorization_tok

            return new FilesystemAdapter(
                new Filesystem($adapter, $config),
                $adapter,
                $config
            );
        });
    }
}
```

Once registered, use the `'dropbox'` driver in `filesystems.php`.

Hello

Hello

Hello

Email Verification

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. This happens seamlessly because Laravel automatically registers the `Illuminate\Auth\Listeners\SendEmailVerificationNotification` listener for the `Illuminate\Auth\Events\Registered` event.

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. Typically, this is included in Laravel's default `0001_01_01_000000_create_users_table.php` database migration.

Routing

To properly implement email verification, three routes will need to be defined. First, a route to display a notice to the user to click the verification link in the email Laravel sent after registration.

Second, a route to handle requests when the user clicks the verification link.

Third, a route to resend the verification email if the user lost the first.

The Email Verification Notice

A route should be defined that returns a view instructing the user to click the verification link emailed to them by Laravel after registration. This view will be shown when users try to access other parts of the application without verifying their email.

The route must be named `verification.notice`:

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

When implementing email verification yourself, you need to define the verification notice view content yourself. For scaffolding including all authentication and verification views, check out the Laravel starter kits.

The Email Verification Handler

Define a route that handles requests when users click the email verification link. It should be named `verification.verify` and include the `auth` and `signed` middlewares:

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;

Route::get('/email/verify/{id}/{hash}', function (EmailVerificationRequest $request) {
    $request->fulfill();

    return redirect('/home');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

This route uses `EmailVerificationRequest`, a form request that Laravel provides, which validates the `id` and `hash` parameters automatically.

Calling `$request->fulfill()` marks the email as verified and dispatches the `Illuminate\Auth\Events\Verified` event. After verification, redirect users as desired.

Resending the Verification Email

If a user misplaces or accidentally deletes the verification email, you can allow the user to request resending it via a route:

```
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

Use the `verified` route middleware to only allow verified users to access certain routes. Laravel's `verified` middleware, alias of `Illuminate\Auth\Middleware\EnsureEmailIsVerified`, is registered automatically:

```
Route::get('/profile', function () {
    // Only verified users may access this route...
})->middleware(['auth', 'verified']);
```

Unverified users attempting to access such routes will be redirected to the verification notice route.

Customization

Verification Email Customization

You can customize the email verification notification message. Use the `toMailUsing` method in your `AppServiceProvider`'s `boot` method, which accepts a closure with `$notifiable` and `$url`:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * Bootstrap any application 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, see the mail notification documentation.

Events

Laravel dispatches an `Illuminate\Auth\Events\Verified` event during the email verification process when using the starter kits. If handling email verification manually, you may dispatch this event yourself upon successful verification.

Encryption

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 cannot 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.

Gracefully Rotating Encryption Keys

If you change your application's encryption key, all authenticated user sessions will be logged out of your application. This is because every cookie, including session cookies, are encrypted by Laravel. In addition, it will no longer be possible to decrypt any data that was encrypted with your previous encryption key.

To mitigate this issue, Laravel allows you to list your previous encryption keys in your application's `APP_PREVIOUS_KEYS` environment variable. This variable may contain a comma-delimited list of all of your previous encryption keys:

```
APP_KEY="base64:J63qRTDLub5NuZvP+kb8YIorGS6qFYHKVo6u7179stY="
APP_PREVIOUS_KEYS="base64:2nLsGFGzyoae2ax3EF2Lyq/hH6QghBGLIq5uL+Gp8/w="
```

When you set this environment variable, Laravel will always use the "current" encryption key when encrypting values. However, when decrypting values, Laravel will first try the current key, and if decryption fails using the current key, Laravel will try all previous keys until one of the keys is able to decrypt the value.

This approach to graceful decryption allows users to keep using your application uninterrupted even if your encryption key is rotated.

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 cannot 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

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

By default, Laravel uses the `bcrypt` hashing driver when hashing data. However, several other hashing drivers are supported, including Argon and Argon2id.

You may specify your application's hashing driver using the `HASH_DRIVER` environment variable. To customize all of Laravel's hashing driver options, you should publish the complete `hashing` configuration file using the Artisan command:

```
php artisan config:publish hashing
```

or

```
php artisan config:publish hashing
```

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 perform this check during the authentication process:

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

Hash Algorithm Verification

To prevent hash algorithm manipulation, Laravel's `Hash::check` method will first verify if the given hash was generated using the application's selected hashing algorithm. If the algorithms are different, a

`RuntimeException` will be thrown.

This is the expected behavior for most applications, where the hashing algorithm is not expected to change and different algorithms can indicate malicious activity. However, if you need to support multiple hashing algorithms within your application, such as when migrating from one algorithm to another, you can disable hash algorithm verification by setting the `HASH_VERIFY` environment variable to `false`:

```
HASH_VERIFY=false
```

Resetting Passwords

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.

Configuration

Your application's password reset configuration file is stored at `config/auth.php`. Be sure to review the options available to you in this file. By default, Laravel is configured to use the `database` password reset driver.

The `driver` configuration option defines where password reset data will be stored. Laravel includes two drivers:

- `database` - password reset data is stored in a relational database.
- `cache` - password reset data is stored in one of your cache-based stores.

Driver Prerequisites

Database

When using the default `database` driver, a table must be created to store your application's password reset tokens. Typically, this is included in Laravel's default `0001_01_01_000000_create_users_table.php` database migration.

Cache

There is also a cache driver available for handling password resets, which does not require a dedicated database table. Entries are keyed by the user's email address, so ensure you are not using email addresses as a cache key elsewhere in your application:

```
'passwords' => [
    'users' => [
        'driver' => 'cache',
        'provider' => 'users',
        'store' => 'passwords', // Optional...
        'expire' => 60,
        'throttle' => 60,
    ],
],
```

To prevent a call to `artisan cache:clear` from flushing your password reset data, you can optionally specify a separate cache store with the `store` configuration key. The value should correspond to a store configured in your `config/cache.php` configuration.

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 created with new Laravel applications.

Next, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. The default user model included with the framework already implements this interface and uses the `Illuminate\Auth\Passwords\CanResetPassword` trait.

Configuring Trusted Hosts

By default, Laravel responds to all requests regardless of the HTTP `Host` header. The `Host` header's value is used when generating absolute URLs during a web request.

Generally, you should configure your web server (like Nginx or Apache) to only send requests matching your hostname. If you cannot modify your web server directly and want Laravel to only respond to certain hostnames, use the `trustHosts` middleware method in your `bootstrap/app.php`. This is especially important for password reset functionality.

Learn more in the TrustHosts middleware documentation.

Routing

To enable users to reset their passwords, several routes need to be defined:

- Routes for requesting a password reset link via email.
- Routes for handling the actual password reset after visiting the emailed link.

Requesting the Password Reset Link

The Password Reset Link Request Form

Define a route that returns the password reset request form view:

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

This view should contain a form with an `email` field for requesting a password reset link.

Handling the Form Submission

Next, define a route to handle the form submission, validate input, and send the reset link:

```
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 $status === Password::ResetLinkSent
        ? back()->with(['status' => __($status)])
```

```
        : back()->withErrors(['email' => __($status)]);
})->middleware('guest')->name('password.email');
```

Laravel's password broker will handle retrieving the user by email and sending the reset link via notifications.

Resetting the Password

The Password Reset Form

Define a route that displays the password reset form when the user clicks the reset link:

```
Route::get('/reset-password/{token}', function (string $token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

This form should include fields for `email`, `password`, `password_confirmation`, and a hidden `token`.

Handling the Reset Form Submission

Define a route that processes the password reset:

```
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');
```

Note: Laravel's `Password` facade is used to validate and process the password reset request.

Deleting Expired Tokens

Password reset tokens stored in the database can be cleared with:

```
php artisan auth:clear-resets
```

To automate this, add it to your task scheduler:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('auth:clear-resets')->everyFifteenMinutes();
```

Customization

Reset Link Customization

You can customize the generated reset link URL using `ResetPassword::createUrlUsing()`:

```
use App\Models\User;
use Illuminate\Auth\Notifications\ResetPassword;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    ResetPassword::createUrlUsing(function (User $user, string $token) {
        return 'https://example.com/reset-password?token=' . $token;
    });
}
```

Reset Email Customization

Override the `sendPasswordResetNotification()` method in your `App\Models\User` model:

```
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));
}
```

Database: Getting Started

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 10.3+ (Version Policy)
- MySQL 5.7+ (Version Policy)
- PostgreSQL 10.0+ (Version Policy)
- SQLite 3.26.0+
- SQL Server 2017+ (Version Policy)

Additionally, MongoDB is supported via the `mongodb/laravel-mongodb` package, which is officially maintained by MongoDB. Check out the Laravel MongoDB documentation for more information.

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
```

By default, foreign key constraints are enabled for SQLite connections. If you would like to disable them, you should set the `DB_FOREIGN_KEYS` environment variable to `false`:

```
DB_FOREIGN_KEYS=false
```

If you use the Laravel installer to create your Laravel application and select SQLite as your database, Laravel will automatically create a `database/database.sqlite` file and run the default database migrations for you.

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@127.0.0.1:3306/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 `DB_URL`) 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,
    'database' => env('DB_DATABASE', 'laravel'),
    'username' => env('DB_USERNAME', 'root'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => env('DB_CHARSET', 'utf8mb4'),
    'collation' => env('DB_COLLATION', 'utf8mb4_unicode_ci'),
    'prefix' => '',
    'prefix_indexes' => true,
    'strict' => true,
    'engine' => null,
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
    ]) : [],
],
```

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 `196.168.1.3` 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 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:

```
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:

```
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` 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 can be useful for logging or debugging. Register your query listener closure in the `boot` method of a service provider:

```php
<?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;
            // $query->toRawSql();
        });
    }
}
```

Monitoring Cumulative Query Time

A common performance bottleneck of modern web applications is the amount of time they spend querying databases. Laravel can invoke a closure or callback when it spends too much time querying the database during a single request.

To get started, provide a query time threshold (in milliseconds) and callback to the `whenQueryingForLongerThan` method. You may invoke this in the `boot` method of a service provider:

```php
<?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, QueryExec
            // 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 re-thrown. If the closure executes successfully, the transaction will be committed automatically. You don't need to manually control rollback or commit:

```
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 that defines how many times a transaction should be retried upon a deadlock. Once attempts are 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 want to control transactions manually, you can begin a transaction with `beginTransaction`, rollback with `rollBack`, and commit with `commit`:

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

DB::beginTransaction();

try {
    // ... perform database operations
```

```
    DB::commit();
} catch (\Exception $e) {
    DB::rollBack();
    throw $e;
}
```

Connecting to the Database CLI

To connect to your database's CLI, use the `db` Artisan command:

```
php artisan db
```

You can specify a connection name to connect to a specific database connection:

```
php artisan db mysql
```

Inspecting Your Databases

Use the `db:show` and `db:table` Artisan commands to get details about your database:

- Show an overview of your database, including size, type, number of open connections, and tables:

```
php artisan db:show
```

- Inspect a specific table's structure:

```
php artisan db:table users
```

You can specify the connection with the `--database` option:

```
php artisan db:show --database=pgsql
```

For large databases, you can include row counts and view details with `--counts` and `--views`. Remember, this may be slow:

```
php artisan db:show --counts --views
```

Monitoring Your Databases

Use the `db:monitor` Artisan command to dispatch an `Illuminate\Database\Events\DatabaseBusy` event if your database manages more than a specified number of open connections:

```
php artisan db:monitor --databases=mysql,pgsql --max=100
```

Schedule this command to run every minute. When the connection count exceeds your threshold, the `DatabaseBusy` event is dispatched, and you should listen for it to send notifications.

Example listener in your `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;
use Illuminate\Database\Events\DatabaseBusy;
```

```
use App\Notifications\DatabaseApproachingMaxConnections;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Event::listen(function (DatabaseBusy $event) {
            Notification::route('mail', 'admin@example.com')
                ->notify(new DatabaseApproachingMaxConnections(
                    $event->connectionName,
                    $event->connections
                ));
        });
    }
}
```

Hello

Database: Pagination

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

If you are using Laravel's default Tailwind pagination views with Tailwind 4.x, your application's `resources/css/app.css` file will already be properly configured to **@source** Laravel's pagination views:

```
@import 'tailwindcss';

@source '../../vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.bl
```

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 inserted into links generated by the paginator.

In this example, the only argument passed to `paginate` is the number of items to display per page (15):

```
<?php

namespace App\Http\Controllers;

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 before retrieving the records. If you do not plan to show the total number of pages, you can use `simplePaginate`, which performs a more efficient query:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You can paginate `Eloquent` queries similarly:

```
use App\Models\User;

$users = User::paginate(15);
```

You can also chain constraints such as `where`:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

Similarly, for `simplePaginate`:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

And for cursor pagination:

```
$users = User::where('votes', '>', 100)->cursorPaginate(15);
```

Multiple Paginator Instances per Page

If rendering multiple paginators on a single page, they should use different query string parameters to avoid conflict. You can specify the page parameter name:

```
use App\Models\User;

$users = User::where('votes', '>', 100)->paginate(15, ['*'], 'users');
```

Cursor Pagination

Cursor pagination improves efficiency for large datasets by constructing `where` clauses based on ordered columns, avoiding the `offset` query's performance issues. It requires an `"order by"` clause on the query and columns that are unique.

Example:

```
$users = DB::table('users')->orderBy('id')->cursorPaginate(15);
```

The URL will include an encoded "cursor" string representing the pagination position, e.g.,:

```
http://localhost/users?cursor=eyJpZCI6MTUsIl9wb2ludHNUb05leHRJdGVtcyI6dHJ1ZX0
```

Note: The query must have an `order by` clause on a unique column for cursor pagination.

Cursor vs. Offset Pagination

Example SQL for second page (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;
```

Advantages of cursor pagination:

- Better performance on large datasets if columns are indexed.
- More consistent results with frequent writes (avoiding skipping or duplication).

Limitations of cursor pagination:

- Only supports "Next" and "Previous" links.
- Requires ordering by a unique column.
- Cannot handle query expressions with parameters or complex expressions in `order by`.

Manually Creating a Paginator

You can create a paginator instance manually with items in memory:

```
use Illuminate\Pagination\Paginator;
use Illuminate\Pagination\LengthAwarePaginator;
use Illuminate\Pagination\CursorPaginator;

$items = [...]; // Your array of items
$paginator = new LengthAwarePaginator($items, $totalCount, $perPage, $currentPage);
```

Note: When manually creating, **slice** your array accordingly, e.g., using `array_slice()`.

Customizing Pagination URLs

Set the base URL of pagination links:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);
    $users->withPath('/admin/users');
    // ...
});
```

Appending Query String Values

Append additional query parameters to pagination links:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);
    $users->appends(['sort' => 'votes']);
    // ...
});
```

Or, include all current query string values:

```
$users = User::paginate(15)->withQueryString();
```

Appending Hash Fragments

Append a hash fragment (anchor) to links:

```
$users = User::paginate(15)->fragment('users');
```

Displaying Pagination Results

When calling `paginate`, you'll get an `Illuminate\Pagination\LengthAwarePaginator` instance, which is iterable and renders links via the `links()` method. Example in Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach

    {{ $users->links() }}
</div>
```

Adjusting the Pagination Link Window

Control how many page links are shown around the current page:

```
{{ $users->onEachSide(5)->links() }}
```

Converting Results to JSON

Since paginator classes implement `Jsonable`, you can return them directly from routes/controllers, and they'll include meta info:

```
use App\Models\User;

Route::get('/users', function () {
    return User::paginate();
});
```

Sample JSON output:

```
{
  "total": 50,
  "per_page": 15,
  "current_page": 1,
  "last_page": 4,
  "current_page_url": "http://laravel.app?page=1",
  "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": [
    // Records...
  ]
}
```

Customizing the Pagination View

To override default views, publish the pagination views:

```
php artisan vendor:publish --tag=laravel-pagination
```

This copies views to `resources/views/vendor/pagination`.

Set default views in `App\Providers\AppServiceProvider`:

```
use Illuminate\Pagination\Paginator;

public function boot(): void
{
    Paginator::defaultView('view-name');
    Paginator::defaultSimpleView('view-name');
}
```

Using Bootstrap

Laravel provides Bootstrap-compatible pagination views. Enable them in `AppServiceProvider`:

```
use Illuminate\Pagination\Paginator;

public function boot(): void
{
    Paginator::useBootstrapFive();
    Paginator::useBootstrapFour();
}
```

Paginator / LengthAwarePaginator Instance Methods

Method	Description
`$paginator->count()`	Get number of items on current page
`$paginator->currentPage()`	Current page number
`$paginator->firstItem()`	Result number of first item
`$paginator->getOptions()`	Get options
`$paginator->getUrlRange($start, $end)`	Generate URL range
`$paginator->hasPages()`	Are there multiple pages?
`$paginator->hasMorePages()`	More items remaining?
`$paginator->items()`	Items for current page
`$paginator->lastItem()`	Result number of last item
`$paginator->lastPage()`	Last page number (not in `simplePaginate`)
`$paginator->nextPageUrl()`	URL for next page
`$paginator->onFirstPage()`	Is on first page?
`$paginator->onLastPage()`	Is on last page?
`$paginator->perPage()`	Items per page
`$paginator->previousPageUrl()`	URL for previous page
`$paginator->total()`	Total items (not in `simplePaginate`)
`$paginator->url($page)`	URL for page number
`$paginator->getPageName()`	Page query variable
`$paginator->setPageName($name)`	Set page query variable
`$paginator->through($callback)`	Transform items with callback

Cursor Paginator Instance Methods

Method	Description
`$paginator->count()`	Number of items on current page
`$paginator->cursor()`	Current cursor object
`$paginator->getOptions()`	Get options
`$paginator->hasPages()`	Are multiple pages?
`$paginator->hasMorePages()`	More items in dataset?
`$paginator->getCursorName()`	Cursor query variable
`$paginator->items()`	Items for current page
`$paginator->nextCursor()`	Cursor for next set
`$paginator->nextPageUrl()`	URL for next page
`$paginator->onFirstPage()`	Is on first page?
`$paginator->onLastPage()`	Is on last page?
`$paginator->perPage()`	Items per page
`$paginator->previousCursor()`	Cursor for previous set
`$paginator->previousPageUrl()`	URL for previous page
`$paginator->setCursorName()`	Set cursor variable
`$paginator->url($cursor)`	URL for specific cursor

Database: Migrations

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 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 can determine the table name from the migration name, it will pre-fill the migration with that table. Otherwise, you can specify the table manually inside the migration.

To specify a custom path for the migration, use the `--path` option:

```
php artisan make:migration create_flights_table --path=custom/path
```

Migration stubs can be customized via stub publishing.

Squashing Migrations

As your application grows, the number of migrations can become large. You can "squash" migrations into a single SQL file with:

```
php artisan schema:dump
```

or with pruning:

```
php artisan schema:dump --prune
```

This writes a schema file to your `database/schema` directory, corresponding to your current database state. Laravel will run the schema's SQL statements before ineligible migrations during subsequent `migrate` commands.

For testing with different databases, dump schemas for each connection:

```
php artisan schema:dump --database=testing --prune
```

Commit the schema files to source control for quick initial setups.

Migration squashing is supported on MariaDB, MySQL, PostgreSQL, and SQLite using their CLI tools.

Migration Structure

Each migration class has two methods: `up()` and `down()`.

- The `up()` method adds new tables, columns, or indexes.
- The `down()` method reverses these operations.

Example 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
{
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::drop('flights');
    }
};
```

Setting the Migration Connection

Specify a custom database connection with `$connection` property:

```
protected $connection = 'pgsql';
```

Skipping Migrations

Define `shouldRun()` to conditionally skip migration:

```
public function shouldRun(): bool
{
    return Feature::active(Flights::class);
}
```

Running Migrations

Execute migrations:

```
php artisan migrate
```

Check migration status:

```
php artisan migrate:status
```

Simulate SQL without executing:

```
php artisan migrate --pretend
```

Isolating Migration Execution

Use `--isolated` to prevent multiple servers from migrating simultaneously, which uses locks:

```
php artisan migrate --isolated
```

Forcing Migrations in Production

Bypass prompts with:

```
php artisan migrate --force
```

Rolling Back Migrations

Rollback last batch:

```
php artisan migrate:rollback
```

Rollback specific steps:

```
php artisan migrate:rollback --step=5
```

Rollback specific batch:

```
php artisan migrate:rollback --batch=3
```

Simulate rollback:

```
php artisan migrate:rollback --pretend
```

Reset all migrations:

```
php artisan migrate:reset
```

Rollback and Migrate in One Command

```
php artisan migrate:refresh
# Or with seed
php artisan migrate:refresh --seed
```

Rollback limited steps:

```
php artisan migrate:refresh --step=5
```

Drop All Tables and Migrate

```
php artisan migrate:fresh
# or with seed
php artisan migrate:fresh --seed
```

Specify database connection:

```
php artisan migrate:fresh --database=admin
```

Note: Drops all tables from the connection, use cautiously.

Tables

Creating Tables

Use `Schema::create`:

```
Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

Determining Table / Column Existence

Check existence:

```
if (Schema::hasTable('users')) {
   // ...
}

if (Schema::hasColumn('users', 'email')) {
   // ...
}
```

Database Connection and Table Options

Use `connection()`:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

Set engine, charset, collation:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine('InnoDB');
    $table->charset('utf8mb4');
    $table->collation('utf8mb4_unicode_ci');
});
```

Create temporary table:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();
});
```

Add comment:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->comment('Business calculations');
});
```

Updating Tables

Use `Schema::table`:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Renaming / Dropping Tables

Rename:

```
Schema::rename($from, $to);
```

Drop:

```
Schema::drop('users');
Schema::dropIfExists('users');
```

Renaming Tables with Foreign Keys

Ensure foreign keys have explicit names before renaming to avoid constraint issues.

Columns

Creating Columns

Use `Schema::table` with `Blueprint`:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Available Column Types

Supported column methods include:

- Boolean: `boolean()`
- String/Text: `char()`, `longText()`, `mediumText()`, `string()`, `text()`, `tinyText()`
- Numeric: `bigIncrements()`, `bigInteger()`, `decimal()`, `double()`, `float()`, `id()`, `increments()`, `integer()`, `mediumIncrements()`, `mediumInteger()`, `smallIncrements()`, `smallInteger()`, `tinyIncrements()`, `tinyInteger()`, etc.
- Date & Time: `dateTime()`, `dateTimeTz()`, `date()`, `time()`, `timeTz()`, `timestamp()`, `timestamps()`, `timestampsTz()`, `softDeletes()`, `softDeletesTz()`, `year()`
- Binary: `binary()`
- JSON: `json()`, `jsonb()`
- UUID/ULID: `ulid()`, `uuid()`, etc.
- Spatial: `geography()`, `geometry()`

Examples:

```
$table->bigIncrements('id');
$table->string('name', 100);
$table->decimal('amount', 8, 2);
$table->json('options');
```

Column Modifiers

Chain methods to modify columns:

```
$table->string('email')->nullable();
$table->string('address')->after('name');
$table->unsignedBigInteger('user_id')->constrained();
$table->foreignId('user_id')->constrained();
```

Modifiers include:

- `->after('column')`
- `->autoIncrement()`
- `->charset('utf8mb4')`
- `->collation('utf8mb4_unicode_ci')`
- `->comment('description')`
- `->default($value)`
- `->first()`
- `->invisible()`
- `->nullable()`
- `->storedAs($expression)`
- `->unsigned()`
- `->useCurrent()`
- `->useCurrentOnUpdate()`
- etc.

Default Expressions

Use `new Expression()` for database-specific defaults, e.g.:

```
$table->json('movies')->default(new Expression('(JSON_ARRAY())'));
```

Column Order

Add column after an existing column:

```
$table->after('password', function (Blueprint $table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Modifying Columns

Change existing column:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

Must specify all attributes you want to keep explicitly.

Renaming Columns

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

Dropping Columns

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
// Multiple columns:
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

Alias Commands for Dropping Common Columns

- `$table->dropMorphs('morphable');`
- `$table->dropRememberToken();`
- `$table->dropSoftDeletes();`
- `$table->dropTimestamps();`
- etc.

Indexes

Creating Indexes

Create unique, index, fulltext, spatial indexes:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
    $table->index('state');
    $table->fullText('body');
    $table->fullText('body')->language('english');
});
```

Chain methods:

```
$table->unique('email');
$table->index(['account_id', 'created_at']);
```

Specify index name:

```
$table->unique('email', 'unique_email');
```

Supported Index Types

- Primary: `$table->primary('id');`
- Composite: `$table->primary(['id', 'parent_id']);`
- Unique: `$table->unique('email');`
- Index: `$table->index('state');`
- Full text: `$table->fullText('body');`, with language
- Spatial: `$table->spatialIndex('location');`

Renaming Indexes

```
$table->renameIndex('from', 'to');
```

Dropping Indexes

Use `dropPrimary()`, `dropUnique()`, `dropIndex()`:

```
$table->dropPrimary('users_id_primary');
$table->dropUnique('users_email_unique');
$table->dropIndex('geo_state_index');
```

Drop multiple columns index:

```
$table->dropIndex(['state']); // index name is generated
```

Foreign Keys

Add foreign key:

```
Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');
    $table->foreign('user_id')->references('id')->on('users');
});
```

Simpler syntax:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

Specify custom table or name:

```
$table->foreignId('user_id')->constrained('users', 'posts_user_id');
```

Set "on delete" / "on update" actions:

```
$table->foreignId('user_id')
    ->constrained()
    ->onUpdate('cascade')
    ->onDelete('cascade');
```

Use fluent methods for actions:

- `cascadeOnUpdate()`
- `restrictOnUpdate()`
- `nullOnUpdate()`
- `noActionOnUpdate()`
- `cascadeOnDelete()`
- etc.

Order of modifiers must be before `constrained()`:

```
$table->foreignId('user_id')->nullable()->constrained();
```

Dropping Foreign Keys

By constraint name:

```
$table->dropForeign('posts_user_id_foreign');
```

By column name:

```
$table->dropForeign(['user_id']);
```

Toggling Foreign Key Constraints

Enable/disable within migrations:

```
Schema::enableForeignKeyConstraints();
Schema::disableForeignKeyConstraints();
Schema::withoutForeignKeyConstraints(function () {
    // constraints are disabled within this closure
});
```

Note: SQLite disables foreign keys by default; enable in config.

Events

Migration events extend `Illuminate\Database\Events\MigrationEvent`:

- `MigrationsStarted`
- `MigrationsEnded`
- `MigrationStarted`
- `MigrationEnded`
- `NoPendingMigrations`
- `SchemaDumped`
- `SchemaLoaded`

Database: Seeding - Laravel 12.x - The PHP Framework For Web Artisans

---

Table of Contents

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 — using the query builder or Eloquent model factories.

Example of modifying the default `DatabaseSeeder` class:

```
<?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

Instead of manually specifying attributes, you can use model factories to generate large amounts of related data. First, review the factory documentation to define factories.

Example: creating 50 users each with 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 can use the `call` method to execute other seed classes:

```
$this->call([
    UserSeeder::class,
    PostSeeder::class,
    CommentSeeder::class,
]);
```

Muting Model Events

While running seeds, you might want to prevent models from dispatching events. Use the `WithoutModelEvents` trait:

```
<?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

Execute the `db:seed` Artisan command to seed your database. By default, it runs the `DatabaseSeeder` class, which can invoke others. Use the `--class` option to run a specific seeder:

```
php artisan db:seed
php artisan db:seed --class=UserSeeder
```

You may also run the `migrate:fresh` command with the `--seed` option, which drops all tables and reruns migrations:

```
php artisan migrate:fresh --seed
php artisan migrate:fresh --seed --seeder=UserSeeder
```

Forcing Seeders to Run in Production

Some operations may alter or delete data. To prevent accidental execution in production, Laravel prompts for confirmation unless you specify the `--force` flag:

```
php artisan db:seed --force
```

Hello

MongoDB

Introduction

MongoDB is one of the most popular NoSQL document-oriented databases, used for its high write load (useful for analytics or IoT) and high availability (easy to set replica sets with automatic failover). It can also shard the database easily for horizontal scalability and has a powerful query language for doing aggregation, text search or geospatial queries.

Instead of storing data in tables of rows or columns like SQL databases, each record in a MongoDB database is a document described in BSON, a binary representation of the data. Applications can then retrieve this information in a JSON format. It supports a wide variety of data types, including documents, arrays, embedded documents, and binary data.

Before using MongoDB with Laravel, we recommend installing and using the `mongodb/laravel-mongodb` package via Composer. The `laravel-mongodb` package is officially maintained by MongoDB, and while MongoDB is natively supported by PHP through the MongoDB driver, the Laravel MongoDB package provides a richer integration with Eloquent and other Laravel features:

```
composer require mongodb/laravel-mongodb
```

Installation

MongoDB Driver

To connect to a MongoDB database, the `mongodb` PHP extension is required. If you are developing locally using Laravel Herd or installed PHP via `php.new`, you already have this extension installed on your system. However, if you need to install the extension manually, you may do so via PECL:

```
pecl install mongodb
```

For more information on installing the MongoDB PHP extension, check out the MongoDB PHP extension installation instructions.

Starting a MongoDB Server

The MongoDB Community Server can be used to run MongoDB locally and is available for installation on Windows, macOS, Linux, or as a Docker container. To learn how to install MongoDB, please refer to the official MongoDB Community installation guide.

The connection string for the MongoDB server can be set in your `.env` file:

```
MONGODB_URI="mongodb://localhost:27017"
MONGODB_DATABASE="laravel_app"
```

For hosting MongoDB in the cloud, consider using MongoDB Atlas. To access a MongoDB Atlas cluster locally from your application, you will need to add your own IP address in the cluster's network settings to the project's IP Access List.

The connection string for MongoDB Atlas can also be set in your `.env` file:

```
MONGODB_URI="mongodb+srv://<username>:<password>@<cluster>.mongodb.net/<dbname>?retryW
MONGODB_DATABASE="laravel_app"
```

Install the Laravel MongoDB Package

Finally, use Composer to install the Laravel MongoDB package:

```
composer require mongodb/laravel-mongodb
```

This package installation will fail if the `mongodb` PHP extension is not installed. The PHP configuration can differ between the CLI and the web server, so ensure the extension is enabled in both configurations.

Configuration

You can configure your MongoDB connection via your application's `config/database.php` file. Add a `mongodb` connection that utilizes the `mongodb` driver:

```
'connections' => [
    'mongodb' => [
        'driver' => 'mongodb',
        'dsn' => env('MONGODB_URI', 'mongodb://localhost:27017'),
        'database' => env('MONGODB_DATABASE', 'laravel_app'),
    ],
],
```

Features

Once your configuration is complete, you can utilize the `mongodb` package and database connection in your application to leverage various powerful features:

- Using Eloquent: Models can be stored in MongoDB collections. The package supports embedded relationships and direct access to the MongoDB driver for raw queries and aggregation pipelines.
- Write complex queries: Use the query builder for advanced queries.
- The MongoDB cache driver is optimized to leverage MongoDB features like TTL indexes to automatically clear expired cache entries.
- Dispatch and process queued jobs with the MongoDB queue driver.
- Storing files in GridFS, via the GridFS Adapter for Flysystem.
- Most third-party packages that use a database connection or Eloquent can be compatible with MongoDB.

To learn more on how to effectively use MongoDB with Laravel, refer to the MongoDB Quick Start guide.

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Eloquent: API Resources

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, use the `make:resource` Artisan command. By default, resources are placed in `app/Http/Resources`. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource UserResource
```

Resource Collections

In addition to individual model transformations, you can generate resource classes responsible for transforming collections of models. This allows your JSON responses to include links and meta information relevant to an entire collection.

Create a collection resource with the `--collection` flag or by including the word `Collection` in the resource name. Collection resources extend `Illuminate\Http\Resources\Json\ResourceCollection`:

```
php artisan make:resource User --collection
php artisan make:resource UserCollection
```

Concept Overview

A resource class represents a single model to be transformed into JSON. Here's a simple example:

```
<?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,
        ];
    }
}
```

The `toArray` method returns an array of attributes to be serialized into JSON. You can access model properties directly via `$this`.

Returning Resources From Routes

Resources are returned by instantiating the resource class with a model instance:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});
```

Using `toResource()` Method

For convenience, models can implement a `toResource()` method, which Laravel will automatically use to discover the appropriate resource:

```
return User::findOrFail($id)->toResource();
```

```
// The resource class, e.g., User, should define:
public function toResource()
{
    return new UserResource($this);
}
```

Resource Collections

When returning a collection or paginated response, use the `collection()` method:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Alternatively, use the collection's `toResourceCollection` method, which Laravel auto-discovers:

```
return User::all()->toResourceCollection();
```

Custom Resource Collections

To include custom meta data, create your own resource collection class:

```
<?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',
            ],
        ];
    }
}
```

Return it from routes/controllers:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Custom collections can override `with()` to add additional meta data:

```
public function with(Request $request): array
{
    return [
        'meta' => [
            'key' => 'value',
        ],
    ];
}
```

Data Wrapping

Outermost resources are wrapped in a `data` key by default. To disable this globally:

```
namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;
```

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        JsonResource::withoutWrapping();
    }
}
```

This affects only the outermost response. Manually added `data` keys remain unaffected.

Wrapping Nested Resources

You can control how relationships are wrapped. Laravel will never double-wrap resources unintentionally, regardless of nesting.

For example, a collection class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the collection into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return ['data' => $this->collection];
    }
}
```

Data Wrapping and Pagination

Paginated responses are always wrapped in `data`, with `meta` and `links` indicating paginator state, regardless of `withoutWrapping()`:

```
{
  "data": [
    {"id": 1, "name": "Eladio Schroeder Sr.", "email": "example@example.com"},
    {"id": 2, "name": "Liliana Mayert", "email": "example2@example.com"}
  ],
  "links": {...},
  "meta": {...}
}
```

To disable wrapping for paginated responses, override `withResponse()` in your resource:

```
public function withResponse(Request $request, JsonResponse $response): void
{
    $response->headers->set('X-Value', 'True');
}
```

Wrapping Nested Resources

You can wrap relationships in a `data` key globally by defining resource classes that return data within that key.

Laravel prevents double-wrapping even with nested collection classes.

Data Wrapping and Pagination Details

Paginated responses include `meta` and `links`. You can customize them via a `paginationInformation()` method receiving paginator data:

```
public function paginationInformation($request, $paginated, $default)
{
    $default['links']['custom'] = 'https://example.com';
    return $default;
}
```

Conditional Attributes

Include attributes only if certain conditions are met using methods like `when()`:

```
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,
    ];
}
```

`when()` also accepts a closure:

```
'secret' => $this->when($request->user()->isAdmin(), function() {
    return 'secret-value';
}),
```

Methods like `whenHas()`, `whenNotNull()`, and `mergeWhen()` enable conditional inclusion based on attribute presence, nullity, or shared conditions.

Conditional Relationships

Include relationships only if loaded using `whenLoaded()`:

```
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,
    ];
}
```

Relationship Counts

Include counts conditionally:

```
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,
    ];
}
```

Use `whenAggregated()` for aggregate functions like `avg`, `sum`, `min`, `max`:

```
'words_avg' => $this->whenAggregated('posts', 'words', 'avg'),
'words_sum' => $this->whenAggregated('posts', 'words', 'sum'),
```

Conditional Pivot Data

Include pivot table data with `whenPivotLoaded()`:

```
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;
        }),
    ];
}
```

For custom pivot attributes:

```
public function toArray(Request $request): array
{
    return [
        'expires_at' => $this->whenPivotLoaded(new Membership, function() {
            return $this->pivot->expires_at;
        }),
    ];
}
```

Or, if the pivot attribute uses a custom accessor:

```
public function toArray(Request $request): array
{
    return [
```

```
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function
            return $this->subscription->expires_at;
        }),
    ];
}
```

Adding Meta Data

Add custom meta info in `toArray()`:

```
public function toArray(Request $request): array
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

Or add top-level meta data in `with()`:

```
public function with(Request $request): array
{
    return [
        'meta' => [
            'key' => 'value',
        ],
    ];
}
```

You can also add meta info when constructing resources via `additional()`:

```
return User::all()
    ->load('roles')
    ->toResourceCollection()
    ->additional(['meta' => ['key' => 'value']]);
```

Resource Responses

Resources can be returned directly from routes/controllers:

```
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return User::findOrFail($id)->toResource();
});
```

You can modify the outgoing HTTP response using `response()` chaining:

```
use App\Http\Resources\UserResource;

Route::get('/user', function () {
    return User::find(1)
        ->toResource()
        ->response()
```

```
        ->header('X-Value', 'True');
});
```

Alternatively, define a `withResponse()` method within the resource class:

```
public function withResponse(Request $request, JsonResponse $response): void
{
    $response->header('X-Value', 'True');
}
```

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return ['id' => $this->id];
    }

    public function withResponse(Request $request, JsonResponse $response): void
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent: 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, use the `toJson` method. Like `toArray`, the `toJson` method is recursive, so all attributes and relations will be converted to JSON. You can also specify JSON encoding options 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:

```
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:

```
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 `$hidden` property to your model. Attributes listed in the `$hidden` array will not be included in the serialized representation:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for serialization.
     *
     * @var array<string>
     */
    protected $hidden = ['password'];
}
```

To hide relationships, add the relationship's method name to your model's `$hidden` property.

Alternatively, you may use the `$visible` property to define an "allow list" of attributes that should be included in your model's array and JSON representation. All attributes not present in the `$visible` array will be hidden:

```
<?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 wish to make some typically hidden attributes visible on a specific model instance, use the `makeVisible` method:

```
return $user->makeVisible('attribute')->toArray();
```

Similarly, to hide attributes that are typically visible, use the `makeHidden` method:

```
return $user->makeHidden('attribute')->toArray();
```

To override all visible or hidden attributes temporarily, use `setVisible` and `setHidden`:

```
return $user->setVisible(['id', 'name'])->toArray();

return $user->setHidden(['email', 'password', 'remember_token'])->toArray();
```

Appending Values to JSON

Occasionally, you may want to add attributes that do not have a corresponding database column. Define an accessor for the value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\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',
        );
    }
}
```

To ensure the accessor is always included in the model's array and JSON outputs, add its name to the `$appends` property:

```
<?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 added to `$appends`, the attribute will be included in both array and JSON forms, respecting `$visible` and `$hidden` settings.

Appending at Run Time

At runtime, you can instruct a model instance to append additional attributes:

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

Date Serialization

Customizing the Default Date Format

Override the `serializeDate` method to customize date serialization:

```
/**
 * 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

Specify the date format in your model's `$casts` property:

```
protected function casts(): array
{
    return [
        'birthday' => 'date:Y-m-d',
        'joined_at' => 'datetime:Y-m-d H:00',
    ];
}
```

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

- Hello
- World

1. Hello
2. World

Hello World

Testing: Getting Started

Introduction

Laravel is built with testing in mind. In fact, support for testing with Pest and 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* focus on a very small, isolated portion of your code, usually a single method. They do not boot your Laravel application and can't access database or framework services.
- *Feature tests* can test larger portions, including interactions of multiple objects or full HTTP requests, such as to a JSON endpoint.
  Generally, most of your tests should be feature tests as they provide the most confidence that your system functions as intended.

An `ExampleTest.php` is provided in both directories. After installing Laravel, run tests with:

- `vendor/bin/pest`
- `vendor/bin/phpunit`
- `php artisan test`

Environment

When testing, Laravel automatically sets the configuration environment to `testing` as specified in `phpunit.xml`. It also configures the session and cache to use the `array` driver so no data persists during testing.

You can customize environment variables in `phpunit.xml`, but remember to clear config cache with `php artisan config:clear` before testing.

The `.env.testing` Environment File

Create a `.env.testing` file in the root. It will be used instead of `.env` when running tests with Pest, PHPUnit, or Artisan commands with `--env=testing`.

Creating Tests

Use the `make:test` Artisan command. Tests default to `tests/Feature`:

```
php artisan make:test UserTest
```

To create in `tests/Unit`, add `--unit`:

```
php artisan make:test UserTest --unit
```

Test stubs can be customized via stub publishing.

Once generated, define your tests as usual in Pest or PHPUnit. Run with:

```
vendor/bin/pest
vendor/bin/phpunit
```

```
php artisan test
```

Example:

```
<?php

test('basic', function () {
    expect(true)->toBeTrue();
});
```

or in PHPUnit:

```
<?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);
    }
}
```

Call `parent::setUp()` / `parent::tearDown()` if overriding these methods in custom test classes to ensure proper setup/cleanup.

Running Tests

You can run tests via Pest or PHPUnit:

```
./vendor/bin/pest
./vendor/bin/phpunit
```

Or using the Artisan command for verbose output:

```
php artisan test
```

Other arguments for `pest` or `phpunit` can be passed to `php artisan test`, e.g.,

```
php artisan test --testsuite=Feature --stop-on-failure
```

Running Tests in Parallel

To reduce test suite time, install the `brianium/paratest` package:

```
composer require brianium/paratest --dev
```

Run tests in parallel:

```
php artisan test --parallel
```

By default, Laravel creates as many processes as CPU cores. Adjust with `--processes`:

```
php artisan test --parallel --processes=4
```

Note: Some options like `--do-not-cache-result` are unavailable with parallel tests.

Parallel Testing and Databases

Laravel manages creating and migrating a test database per process, suffixed with a token (e.g., `your_db_test_1`, `your_db_test_2`).
To force recreation of test databases:

```
php artisan test --parallel --recreate-databases
```

Parallel Testing Hooks

Use the `ParallelTesting` facade to prepare resources for each test process:

```
<?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
{
    public function boot(): void
    {
        ParallelTesting::setUpProcess(function (int $token) {
            // Setup code for each process
        });

        ParallelTesting::setUpTestCase(function (int $token, TestCase $testCase) {
            // Setup before test case
        });

        ParallelTesting::setUpTestDatabase(function (string $database, int $token) {
            Artisan::call('db:seed');
        });

        ParallelTesting::tearDownTestCase(function (int $token, TestCase $testCase) {
            // Cleanup after test case
        });

        ParallelTesting::tearDownProcess(function (int $token) {
            // Cleanup after process
        });
    }
}
```

Accessing the Parallel Testing Token

Retrieve the current process's token with:

```
$token = ParallelTesting::token();
```

Reporting Test Coverage

Use `--coverage` when running tests:

```
php artisan test --coverage
```

Enforcing Minimum Coverage Threshold

Specify with `--min`:

```
php artisan test --coverage --min=80.3
```

Profiling Tests

Use `--profile` to list the slowest ten tests:

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

Hello

Console Tests

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('console command', function () {
    $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:

```
test('console command', function () {
    $this
        ->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.')
        ->assertExitCode(0);
});
```

You may also assert that a console command does not generate any output using the `doesntExpectOutput` method:

```
$this->artisan('example')
    ->doesntExpectOutput()
    ->assertExitCode(0);
```

The `expectsOutputToContain` and `doesntExpectOutputToContain` methods may be used to make assertions against a portion of the output:

```
test('console command', function () {
    $this
        ->expectsOutputToContain('Taylor')
        ->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, 'user1@example.com'],
        [2, 'user2@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;

    // ...
}
```

On this page

Hello

Database Testing

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

use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('basic example', function () {
    $response = $this->get('/');

    // ...
});
```

Or in a more traditional 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;

test('models can be instantiated', function () {
    $user = User::factory()->create();

    // ...
});
```

Alternatively, in a traditional test method:

```
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

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('orders can be created', function () {
    // 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,
        // ...
    ]);
});
```

Or in a traditional test 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
{
```

```
    /**
     * Indicates whether the default seeder should run before each test.
     *
     * @var bool
     */
    protected $seed = true;
}
```

When the `$seed` property is `true`, the test will run the `Database\Seeders\DatabaseSeeder` class before each test that uses the `RefreshDatabase` trait. However, you can specify a specific seeder that should be executed by defining a `$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 Pest or 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);
```

assertDatabaseEmpty

Assert that a table in the database contains no records:

```
$this->assertDatabaseEmpty('users');
```

assertDatabaseHas

Assert that a table in the database contains records matching the given key / value query constraints:

```
$this->assertDatabaseHas('users', [
    'email' => 'example@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' => 'example@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

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;

test('something can be mocked', function () {
    $this->instance(
        Service::class,
        Mockery::mock(Service::class, function (MockInterface $mock) {
            $mock->expects('process');
        })
    );
});
```

Alternatively, in a class:

```
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->expects('process');
        })
    );
}
```

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->expects('process');
});
```

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->expects('process');
});
```

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 dependency injection.

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 `expects` 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:

```
use Illuminate\Support\Facades\Cache;

test('get index', function () {
    Cache::expects('get')
        ->with('key')
        ->andReturn('value');

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

    // ...
});
```

Alternatively, in a feature test:

```
namespace Tests\Feature;

use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

class UserControllerTest extends TestCase
{
    public function test_get_index(): void
    {
        Cache::expects('get')
            ->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;

test('values are stored in cache', function () {
    Cache::spy();

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

```
    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->with('name', 'Taylor', 10);
});
```

Similarly, in a class:

```
use Illuminate\Support\Facades\Cache;

public function test_values_are_stored_in_cache(): void
{
    Cache::spy();

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

    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->with('name', 'Taylor', 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()`. Laravel's base feature test class includes helpers that allow you to manipulate the current time:

```
test('time can be manipulated', function () {
    // 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();
});
```

Similarly, in a class:

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

All of these methods are primarily useful for testing time-sensitive application behavior, such as locking inactive posts after a certain period:

```
use App\Models\Thread;

test('forum threads lock after one week of inactivity', function () {
    $thread = Thread::factory()->create();

    $this->travel(1)->week();

    expect($thread->isLockedByInactivity())->toBeTrue();
});
```

Similarly in a class:

```
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());
}
```

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 during a specific moment...
});
```

All of the methods discussed above are largely useful for testing application behavior that depends on the current time, such as news articles expiring or posts locking after inactivity.

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

Hello

Hello

Laravel Reverb

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

You may install Reverb using the `install:broadcasting` Artisan command:

```
php artisan install:broadcasting
```

Configuration

Behind the scenes, the `install:broadcasting` Artisan command will run the `reverb:install` command, which will install Reverb with a sensible set of default configuration options. You can customize by updating Reverb's environment variables or by editing the `config/reverb.php` file.

Application Credentials

To establish a connection, Reverb requires application credentials exchanged between client and server. Set these in your environment:

```
REVERB_APP_ID=my-app-id
REVERB_APP_KEY=my-app-key
REVERB_APP_SECRET=my-app-secret
```

Allowed Origins

Define from which origins client requests can originate in `config/reverb.php` under `apps`:

```
'apps' => [
    [
        'app_id' => 'my-app-id',
        'allowed_origins' => ['laravel.com'],
        // ...
    ],
]
```

Use `'*'` to allow all origins:

```
'apps' => [
    [
        'app_id' => 'my-app-id',
        'allowed_origins' => ['*'],
        // ...
    ],
]
```

Additional Applications

Multiple applications can be served using a single Reverb installation by defining multiple entries:

```
'apps' => [
    [
        'app_id' => 'my-app-one',
        // ...
    ],
    [
        'app_id' => 'my-app-two',
        // ...
    ],
],
```

SSL

Secure WebSocket connections may be terminated at your web server (e.g., Nginx). For direct secure connections to Reverb, set the `REVERB_HOST` environment variable or pass the hostname when starting the server:

```
php artisan reverb:start --host="0.0.0.0" --port=8080 --hostname="laravel.test"
```

For certificates, specify `tls` options in `config/reverb.php`:

```
'options' => [
    'tls' => [
        'local_cert' => '/path/to/cert.pem',
    ],
],
```

Running the Server

Start Reverb with:

```
php artisan reverb:start
```

Default is `0.0.0.0:8080`. Customize host/port:

```
php artisan reverb:start --host=127.0.0.1 --port=9000
```

Use environment variables (`REVERB_SERVER_HOST` , `REVERB_SERVER_PORT`) instead of arguments. Remember, `REVERB_HOST` and `REVERB_PORT` are for Laravel broadcast configuration.

Debugging

Enable debug logs with:

```
php artisan reverb:start --debug
```

Restarting

Since Reverb runs persistently, restart with:

```
php artisan reverb:restart
```

Ensure graceful shutdown; for process managers like Supervisor, it will auto-restart.

Monitoring

Reverb can be integrated with Laravel Pulse to track connections/messages:

```
use Laravel\Reverb\Pulse\Recorders\ReverbConnections;
use Laravel\Reverb\Pulse\Recorders\ReverbMessages;

'recorders' => [
    ReverbConnections::class => [
        'sample_rate' => 1,
    ],
    ReverbMessages::class => [
        'sample_rate' => 1,
    ],
]
```

Add Pulse dashboard cards:

```
<x-pulse>
    <livewire:reverb.connections cols="full" />
    <livewire:reverb.messages cols="full" />
    ...
</x-pulse>
```

Run the `pulse:check` daemon to poll updates; required only on one server in scaled deployment.

Running in Production

Optimize your environment for WebSocket load:

- For Forge: enable Reverb in the application panel.
- Manage open files: ensure system limits are high enough using `ulimit -n` and system config (`/etc/security/limits.conf`).

Open Files

Unix limits:

```
ulimit -n
```

Set higher in `/etc/security/limits.conf` :

```
forge   soft  nofile  10000
forge   hard  nofile  10000
```

Event Loop

Uses ReactPHP; defaults to `stream_select` , limited to 1024 connections. Use `ext-uv` for higher concurrency:

```
pecl install uv
```

Web Server

Use a reverse proxy (e.g., Nginx):

```
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;
    }
    ...
}
```

Reverb listens at `/app` for WebSocket and `/apps` for API requests.

Ports

Unix systems restrict ports; check current range:

```
cat /proc/sys/net/ipv4/ip_local_port_range
```

Adjust in `/etc/sysctl.conf` if needed.

Process Management

Use Supervisor:

```
[supervisord]
...
minfds=10000
```

Scaling

Enable horizontal scaling with Redis:

Set environment:

```
REVERB_SCALING_ENABLED=true
```

Ensure a central Redis server; Reverb servers communicate via Redis, distributing messages. Reverb servers can run behind load balancers for high availability.

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello

Hello

Hello

Hello

<!-- The provided content is mostly a large HTML document; converting the entire conte

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Laravel Envoy

Introduction

Laravel Envoy is a tool for executing common tasks you run on your remote servers. Using *Blade* style syntax, you can easily set up 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 Composer:

```
composer require laravel/envoy --dev
```

Once 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 blocks of Envoy. They define shell commands that execute on your remote servers when invoked. For example, you might define a task that runs `php artisan queue:restart` on all queue worker servers.

All Envoy tasks should be defined in an `Envoy.blade.php` file at the root of your application. Here's an example:

```
@servers(['web' => ['user@domain.com'], 'workers' => ['user@domain.com']])

@task('restart-queues', ['on' => 'workers'])
    cd /home/user/example.com
    php artisan queue:restart
@endtask
```

In this example, an array of `@servers` is defined at the top, allowing you to reference these servers via the `on` option of your task declarations. The `@servers` declaration should always be on a single line. Inside your `@task` declarations, place the shell commands to execute when the task is invoked.

Local Tasks

To run a script locally, specify the server's IP as `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Importing Envoy Tasks

You can import other Envoy files to include their stories and tasks:

```
@import('vendor/package/Envoy.blade.php')
```

Multiple Servers

Envoy allows you to run tasks across multiple servers. Add additional servers to your `@servers` declaration with unique names:

```
@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 run serially. To execute in parallel, add the `parallel` option:

```
@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

To execute PHP code before tasks, use the `@setup` directive:

```
@setup
    $now = new DateTime;
@endsetup
```

To include other PHP files before tasks, use `@include`:

```
@include('vendor/autoload.php')
```

Variables

Pass arguments via command line:

```
php vendor/bin/envoy run deploy --branch=master
```

Access options and pass Blade variables:

```
@servers(['web' => ['user@domain.com']])

@task('deploy', ['on' => 'web'])
    cd /home/user/example.com

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate --force
@endtask
```

Stories

Stories group tasks:

```
@servers(['web' => ['user@domain.com']])

@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
```

Invoke the story like a task:

```
php vendor/bin/envoy run deploy
```

Hooks

Envoy supports hooks: `@before`, `@after`, `@error`, `@success`, and `@finished`. These execute PHP code locally and in order.

Before Hooks

Execute before each task:

```
@before
    if ($task === 'deploy') {
        // do something
    }
@endbefore
```

After Hooks

Execute after each task:

```
@after
    if ($task === 'deploy') {
        // do something
    }
@endafter
```

Error Hooks

Execute on task failure:

```
@error
    if ($task === 'deploy') {
        // handle error
    }
@enderror
```

Success Hooks

If no errors occurred:

```
@success
    // success actions
@endsuccess
```

Finished Hooks

After all tasks (regardless of exit code):

```
@finished
    if ($exitCode > 0) {
        // handle errors
    }
@endfinished
```

Running Tasks

Execute a task or story:

```
php vendor/bin/envoy run deploy
```

Confirming Task Execution

Add `confirm` to prompt before execution:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Notifications

Slack

Send Slack notifications:

```
@finished
    @slack('webhook-url', '#channel')
@endfinished
```

Customize message:

```
@finished
    @slack('webhook-url', '#channel', 'Hello, Slack.')
```

```
@endfinished
```

Discord

Send Discord notifications:

```
@finished
    @discord('discord-webhook-url')
@endfinished
```

Telegram

Send Telegram notifications:

```
@finished
    @telegram('bot-id', 'chat-id')
@endfinished
```

Microsoft Teams

Send Teams notifications:

```
@finished
    @microsoftTeams('webhook-url')
@endfinished
```

Laravel Fortify

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 can always interact directly with Laravel's authentication services by following the documentation on authentication, password reset, and email verification.

What is Fortify?

As mentioned above, Laravel Fortify is a frontend agnostic authentication backend for Laravel. It registers routes and controllers for features like login, registration, password reset, email verification, etc.

You are not required to use Fortify to access Laravel's authentication features. Manual interaction is always possible through the official docs.

If you're new to Laravel, consider exploring starter kits that include pre-built UIs with Tailwind CSS, which work seamlessly with Fortify.

Laravel Fortify basically provides backend routes and controllers, and allows you to build your own frontend UI.

When Should I Use Fortify?

- If you are using Laravel's application starter kits, no need to install Fortify—these kits include full authentication scaffolding.
- If you are building an app without starter kits and need authentication features, you can:
  - Implement manually, or
  - Use Fortify for the backend implementation.

Your frontend will make requests to Fortify's routes as detailed here.

Laravel Fortify and Laravel Sanctum

Many developers confuse the difference:

- Laravel Sanctum manages API tokens and authenticates users via session cookies or tokens. It does NOT provide routes for registration, password resets, etc.
- Fortify provides the backend routes for registration, password resets, and email verification.

Using both together is common, especially when building APIs or SPAs. Sanctum manages API tokens, Fortify handles user management.

Installation

1. Install via Composer:

```
composer require laravel/fortify
```

2. Publish Fortify's resources:

```
php artisan fortify:install
```

This publishes actions, configuration, migrations, etc.

3. Migrate your database:

```
php artisan migrate
```

Fortify Features

Define which backend features are enabled in the `config/fortify.php` file via the `features` array. Recommended features include:

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
],
```

Disabling Views

By default, Fortify defines routes that return views like login or registration screens.

To disable views (for example, if building an SPA), set:

```
'views' => false,
```

in your `config/fortify.php`.

Disabling Views and Password Reset

Even if views are disabled, you should still define a route named `password.reset` because Laravel's password reset email links rely on it.

Authentication Views

You need to instruct Fortify how to return your login view. Typically, this is set up in your `FortifyServiceProvider`:

```
use Laravel\Fortify\Fortify;

// Inside your service provider's boot() method
```

```
public function boot(): void
{
    Fortify::loginView(function () {
        return view('auth.login');
    });
}
```

Your login form should POST to `/login` with fields:

- `email` or `username` (matching your config)
- `password`
- optional `remember`

Successful login redirects to the `home` URI; errors are available in `$errors`.

Customizing User Authentication

You can override authentication logic via `Fortify::authenticateUsing()`, e.g.:

```
Fortify::authenticateUsing(function (Request $request) {
    $user = User::where('email', $request->email)->first();

    if ($user && Hash::check($request->password, $user->password)) {
        return $user;
    }
});
```

Authentication Guard

Configure which auth guard to use in `config/fortify.php`, typically `'web'`. Ensure it implements `Illuminate\Contracts\Auth\StatefulGuard`. For SPAs, use the `web` guard with Sanctum.

Customizing the Authentication Pipeline

Fortify authenticates via a pipeline of classes. You can define a custom pipeline with `Fortify::authenticateThrough()`:

```
Fortify::authenticateThrough(function (Request $request) {
    return array_filter([
        config('fortify.limiters.login') ? null : EnsureLoginIsNotThrottled::class,
        config('fortify.lowercase_usernames') ? CanonicalizeUsername::class : null,
        Features::enabled(Features::twoFactorAuthentication()) ? RedirectIfTwoFactorAu
        AttemptToAuthenticate::class,
        PrepareAuthenticatedSession::class,
    ]);
});
```

You can customize this pipeline as needed.

Authentication Throttling

Fortify throttles login attempts with `EnsureLoginIsNotThrottled`. You can customize rate limiting in `config/fortify.php` via `fortify.limiters.login`.

Custom Redirects after Login/Logout

Customize redirect behavior by binding `LoginResponse` or `LogoutResponse` into the service container in your provider:

```
use Laravel\Fortify\Contracts\LogoutResponse;

public function register(): void
{
    $this->app->instance(LogoutResponse::class, new class implements LogoutResponse {
        public function toResponse($request)
        {
            return redirect('/');
        }
    });
}
```

On login, success redirect is configured by `home`; logout redirect defaults to `/`.

Two-Factor Authentication

Enabled via `Features::twoFactorAuthentication()`. Users will input a code generated by an app like Google Authenticator.

Setup

Ensure your `User` model uses `Laravel\Fortify\TwoFactorAuthenticatable`:

```
use Laravel\Fortify\TwoFactorAuthenticatable;

class User extends Authenticatable
{
    use TwoFactorAuthenticatable;
}
```

Managing 2FA

- Show QR code:

```
$request->user()->twoFactorQrCodeSvg();
```

- Confirm 2FA setup: POST to `/user/confirmed-two-factor-authentication` with code.
- Disable: DELETE to `/user/two-factor-authentication`.

Enabling

POST to `/user/two-factor-authentication` to enable 2FA. You will be prompted to scan QR code and confirm via code.

Confirming

Provide a 2FA code to verify setup:

```
$request->user()->twoFactorConfirm($code);
```

Recovery codes can be accessed or regenerated:

```
(array) $request->user()->recoveryCodes();
```

Regenerate with POST to `/user/two-factor-recovery-codes`.

Protecting Routes

Use Laravel's built-in `verified` middleware to enforce email verification:

```
Route::get('/dashboard', function () {
    // ...
})->middleware(['verified']);
```

Password Confirmation

Some actions require re-entering password. Use `Fortify::confirmPasswordView()`:

```
Fortify::confirmPasswordView(function () {
    return view('auth.confirm-password');
});
```

POST to `/user/confirm-password` with `password` field.

(The rest of the page contains navigation, footer, links to products, resources, partners, and social links).

Note: For further customization, implement your own views, actions, and route logic; the above are simple examples to get started.

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Laravel Mix

Introduction

Laravel Mix is a legacy package that is no longer actively maintained. Vite may be used as a modern alternative.

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.

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World