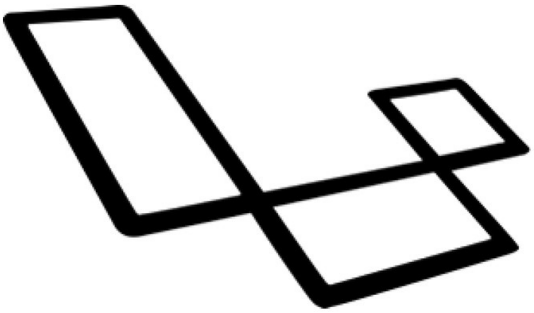


Laravel

8.x

DOCUMENTATION

Laravel Documentation - 8.x

https://laravel.com/docs/

eBook compiled from the source

https://github.com/laravel/docs/

by david@mundosaparte.com

Get the latest version at https://github.com/driade/laravel-book

Date: Tuesday, 29-Dec-20 00:33:54 CET

Contents

Prologue

- Release Notes
- Upgrade Guide
- Contribution Guide

Getting Started

- Installation
- Configuration
- Directory Structure
- Starter Kits
- Deployment

Architecture Concepts

- Request Lifecycle
- Service Container
- Service Providers
- Facades

The Basics

- Routing
- Middleware
- CSRF Protection
- Controllers
- Requests
- Responses
- Views
- Blade Templates
- URL Generation
- Session
- Validation
- Error Handling
- Logging

Digging Deeper

- Artisan Console
- Broadcasting
- Cache
- Collections
- Compiling Assets
- Contracts
- Events
- File Storage
- Helpers
- HTTP Client
- Localization
- Mail
- Notifications
- Package Development
- Queues
- Task Scheduling

Security

- Authentication

Authorization
Email Verification
Encryption
Hashing
Password Reset

Database

Getting Started
Query Builder
Pagination
Migrations
Seeding
Redis

Eloquent ORM

Getting Started
Relationships
Collections
Mutators / Casts
API Resources
Serialization

Testing

Getting Started
HTTP Tests
Console Tests
Browser Tests
Database
Mocking

Packages

Cashier (Stripe)
Cashier (Paddle)
Dusk
Envoy
Fortify
Homestead
Horizon
Passport
Sail
Sanctum
Scout
Telescope
Valet

Prologue

Release Notes

- Versioning Scheme
- Support Policy
- Laravel 8

Versioning Scheme

Laravel and its other first-party packages follow Semantic Versioning. Major framework releases are released every six months (~March and ~September), 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 `^8.0`, 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.

Support Policy

For LTS releases, such as Laravel 6, bug fixes are provided for 2 years and security fixes are provided for 3 years. These releases provide the longest window of support and maintenance. For general releases, bug fixes are provided for 7 months and security fixes are provided for 1 year. For all additional libraries, including Lumen, only the latest release receives bug fixes. In addition, please review the database versions supported by Laravel.

Version	Release	Bug Fixes Until	Security Fixes Until
6 (LTS)	September 3rd, 2019	October 5th, 2021	September 3rd, 2022
7	March 3rd, 2020	October 6th, 2020	March 3rd, 2021
8	September 8th, 2020	April 6th, 2021	September 8th, 2021

Laravel 8

Laravel 8 continues the improvements made in Laravel 7.x by introducing Laravel Jetstream, model factory classes, migration squashing, job batching, improved rate limiting, queue improvements, dynamic Blade components, Tailwind pagination views, time testing helpers, improvements to `artisan serve`, event listener improvements, and a variety of other bug fixes and usability improvements.

Laravel Jetstream

Laravel Jetstream was written by Taylor Otwell.

Laravel Jetstream is a beautifully designed application scaffolding for Laravel. Jetstream provides the perfect starting point for your next project and includes login, registration, email verification, two-factor authentication, session management, API support via Laravel Sanctum, and optional team management. Laravel Jetstream replaces and improves upon the legacy authentication UI scaffolding available for previous versions of Laravel.

Jetstream is designed using Tailwind CSS and offers your choice of Livewire or Inertia scaffolding.

Models Directory

By overwhelming community demand, the default Laravel application skeleton now contains an `app/Models` directory. We hope you enjoy this new home for your Eloquent models! All relevant generator commands have been updated to assume models exist within the `app/Models` directory if it exists. If the directory does not exist, the framework will assume your models should be placed within the `app` directory.

Model Factory Classes

Model factory classes were contributed by [Taylor Otwell](#).

Eloquent model factories have been entirely re-written as class based factories and improved to have first-class relationship support. For example, the `UserFactory` included with Laravel is written like so:

```
<?php

namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = User::class;

    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'name' => $this->faker->name,
            'email' => $this->faker->unique()->safeEmail,
            'email_verified_at' => now(),
            'password' => '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }
}
```

Thanks to the new `HasFactory` trait available on generated models, the model factory may be used like so:

```
use App\Models\User;

User::factory()->count(50)->create();
```

Since model factories are now simple PHP classes, state transformations may be written as class methods. In addition, you may add any other helper classes to your Eloquent model factory as needed.

For example, your `User` model might have a `suspended` state that modifies one of its default attribute values. You may define your state transformations using the base factory's `state` method. You may name your state method anything you like. After all, it's just a typical PHP method:

```
/**
 * Indicate that the user is suspended.
 *
 * @return \Illuminate\Database\Eloquent\Factories\Factory
 */
public function suspended()
{
    return $this->state([
        'account_status' => 'suspended',
    ]);
}
```

After defining the state transformation method, we may use it like so:

```
use App\Models\User;

User::factory()->count(5)->suspended()->create();
```

As mentioned, Laravel 8's model factories contain first class support for relationships. So, assuming our `User` model has a `posts` relationship method, we may simply run the following code to generate a user with three

posts:

```
$users = User::factory()
            ->hasPosts(3, [
                'published' => false,
            ])
            ->create();
```

To ease the upgrade process, the laravel/legacy-factories package has been released to provide support for the previous iteration of model factories within Laravel 8.x.

Laravel's re-written factories contain many more features that we think you will love. To learn more about model factories, please consult the database testing documentation.

Migration Squashing

Migration squashing was contributed by Taylor Otwell.

As you build your application, you may accumulate more and more migrations over time. This can lead to your migration directory becoming bloated with potentially hundreds of migrations. If you're using MySQL or PostgreSQL, you may now "squash" your migrations into a single SQL file. To get started, execute the `schema:dump` command:

```
php artisan schema:dump

// Dump the current database schema and prune all existing migrations...
php artisan schema:dump --prune
```

When you execute this command, Laravel will write a "schema" file to your `database/schema` directory. Now, when you attempt to migrate your database and no other migrations have been executed, Laravel will execute the schema file's SQL first. After executing the schema file's commands, Laravel will execute any remaining migrations that were not part of the schema dump.

Job Batching

Job batching was contributed by Taylor Otwell & Mohamed Said.

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing.

The new `batch` method of the `Bus` facade may be used to dispatch a batch of jobs. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the `then`, `catch`, and `finally` methods to define completion callbacks for the batch. Each of these callbacks will receive an `Illuminate\Bus\Batch` instance when they are invoked:

```
use App\Jobs\ProcessPodcast;
use App\Podcast;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ProcessPodcast(Podcast::find(1)),
    new ProcessPodcast(Podcast::find(2)),
    new ProcessPodcast(Podcast::find(3)),
    new ProcessPodcast(Podcast::find(4)),
    new ProcessPodcast(Podcast::find(5)),
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
    // First batch job failure detected...
})->finally(function (Batch $batch) {
    // The batch has finished executing...
})->dispatch();

return $batch->id;
```

To learn more about job batching, please consult the queue documentation.

Improved Rate Limiting

Rate limiting improvements were contributed by Taylor Otwell.

Laravel's request rate limiter feature has been augmented with more flexibility and power, while still maintaining backwards compatibility with previous release's `throttle` middleware API.

Rate limiters are defined using the `RateLimiter` facade's `for` method. The `for` method accepts a rate limiter name and a closure that returns the limit configuration that should apply to routes that are assigned this rate limiter:

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

RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000);
});
```

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

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100);
});
```

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

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100)->by($request->ip());
});
```

Rate limiters may be attached to routes or route groups using the `throttle` middleware. The throttle middleware accepts the name of the rate limiter you wish to assign to the route:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        //
    });

    Route::post('/video', function () {
        //
    });
});
```

To learn more about rate limiting, please consult the routing documentation.

Improved Maintenance Mode

Maintenance mode improvements were contributed by Taylor Otwell with inspiration from Spatie.

In previous releases of Laravel, the `php artisan down` maintenance mode feature may be bypassed using an "allow list" of IP addresses that were allowed to access the application. This feature has been removed in favor of a simpler "secret" / token solution.

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

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

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

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

When accessing this hidden route, you will then be redirected to the `/` route of the application. Once the cookie

has been issued to your browser, you will be able to browse the application normally as if it was not in maintenance mode.

Pre-Rendering The Maintenance Mode View

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

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

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

Closure Dispatch / Chain `catch`

Catch improvements were contributed by Mohamed Said.

Using the new `catch` method, you may now provide a closure that should be executed if a queued closure fails to complete successfully after exhausting all of your queue's configured retry attempts:

```
use Throwable;

dispatch(function () use ($podcast) {
    $podcast->publish();
})->catch(function (Throwable $e) {
    // This job has failed...
});
```

Dynamic Blade Components

Dynamic Blade components were contributed by Taylor Otwell.

Sometimes you may need to render a component but not know which component should be rendered until runtime. In this situation, you may now use Laravel's built-in `dynamic-component` component to render the component based on a runtime value or variable:

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

To learn more about Blade components, please consult the Blade documentation.

Event Listener Improvements

Event listener improvements were contributed by Taylor Otwell.

Closure based event listeners may now be registered by only passing the closure to the `Event::listen` method. Laravel will inspect the closure to determine which type of event the listener handles:

```
use App\Events\PodcastProcessed;
use Illuminate\Support\Facades\Event;

Event::listen(function (PodcastProcessed $event) {
    //
});
```

In addition, closure based event listeners may now be marked as queueable using the `Illuminate\Events\queueable` function:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;

Event::listen(queueable(function (PodcastProcessed $event) {
    //
```

```
}));
```

Like queued jobs, you may use the `onConnection`, `onQueue`, and `delay` methods to customize the execution of the queued listener:

```
Event::listen(queueable(function (PodcastProcessed $event) {
    //
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10)));
```

If you would like to handle anonymous queued listener failures, you may provide a closure to the `catch` method while defining the `queueable` listener:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
use Throwable;

Event::listen(queueable(function (PodcastProcessed $event) {
    //
})->catch(function (PodcastProcessed $event, Throwable $e) {
    // The queued listener failed...
}));
```

Time Testing Helpers

Time testing helpers were contributed by [Taylor Otwell](#) with inspiration from Ruby on Rails.

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 now includes helpers that allow you to manipulate the current time:

```
public function testTimeCanBeManipulated()
{
    // 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();
}
```

Artisan `serve` Improvements

Artisan `serve` improvements were contributed by [Taylor Otwell](#).

The Artisan `serve` command has been improved with automatic reloading when environment variable changes are detected within your local `.env` file. Previously, the command had to be manually stopped and restarted.

Tailwind Pagination Views

The Laravel paginator has been updated to use the Tailwind CSS framework by default. Tailwind CSS is a highly customizable, low-level CSS framework that gives you all of the building blocks you need to build bespoke designs without any annoying opinionated styles you have to fight to override. Of course, Bootstrap 3 and 4 views remain available as well.

Routing Namespace Updates

In previous releases of Laravel, the `RouteServiceProvider` contained a `$namespace` property. This property's value would automatically be prefixed onto controller route definitions and calls to the `action` helper / `URL::action` method. In Laravel 8.x, this property is `null` by default. This means that no automatic namespace prefixing will be done by Laravel. Therefore, in new Laravel 8.x applications, controller route definitions should be defined using standard PHP callable syntax:

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

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

Calls to the `action` related methods should use the same callable syntax:

```
action([UserController::class, 'index']);

return Redirect::action([UserController::class, 'index']);
```

If you prefer Laravel 7.x style controller route prefixing, you may simply add the `$namespace` property into your application's `RouteServiceProvider`.

> NOTE This change only affects new Laravel 8.x applications. Applications upgrading from Laravel 7.x will still have the `$namespace` property in their `RouteServiceProvider`.

Prologue

Upgrade Guide

- Upgrading To 8.0 From 7.x

High Impact Changes

- Model Factories
- Queue `retryAfter` Method
- Queue `timeoutAt` Property
- Queue `allOnQueue` and `allOnConnection`
- Pagination Defaults
- Seeder & Factory Namespaces

Medium Impact Changes

- PHP 7.3.0 Required
- Failed Jobs Table Batch Support
- Maintenance Mode Updates
- The `php artisan down --message` Option
- The `assertExactJson` Method

Upgrading To 8.0 From 7.x

Estimated Upgrade Time: 15 Minutes

> NOTE We attempt to document every possible breaking change. Since some of these breaking changes are in obscure parts of the framework only a portion of these changes may actually affect your application.

PHP 7.3.0 Required

Likelihood Of Impact: Medium

The new minimum PHP version is now 7.3.0.

Updating Dependencies

Update the following dependencies in your `composer.json` file:

- `guzzlehttp/guzzle` to `^7.0.1`
- `facade/ignition` to `^2.3.6`
- `laravel/framework` to `^8.0`
- `laravel/ui` to `^3.0`
- `nunomaduro/collision` to `^5.0`
- `phpunit/phpunit` to `^9.0`

The following first-party packages have new major releases to support Laravel 8. If applicable, you should read their individual upgrade guides before upgrading:

- Horizon v5.0
- Passport v10.0
- Socialite v5.0
- Telescope v4.0

In addition, the Laravel installer has been updated to support `composer create-project` and Laravel Jetstream. Any installer older than 4.0 will cease to work after October 2020. You should upgrade your global installer to `^4.0` as soon as possible.

Finally, examine any other third-party packages consumed by your application and verify you are using the proper version for Laravel 8 support.

Collections

The `isset` Method

Likelihood Of Impact: Low

To be consistent with typical PHP behavior, the `offsetExists` method of `Illuminate\Support\Collection` has been updated to use `isset` instead of `array_key_exists`. This may present a change in behavior when dealing with collection items that have a value of `null`:

```
$collection = collect([null]);

// Laravel 7.x - true
isset($collection[0]);

// Laravel 8.x - false
isset($collection[0]);
```

Database

Seeder & Factory Namespaces

Likelihood Of Impact: High

Seeders and factories are now namespaced. To accommodate for these changes, add the `Database\Seeders` namespace to your seeder classes. In addition, the previous `database/seeds` directory should be renamed to `database/seeders`:

```
<?php

namespace Database\Seeders;

use App\Models\User;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        ...
    }
}
```

If you are choosing to use the `laravel/legacy-factories` package, no changes to your factory classes are required. However, if you are upgrading your factories, you should add the `Database\Factories` namespace to those classes.

Next, in your `composer.json` file, remove `classmap` block from the `autoload` section and add the new namespaced class directory mappings:

```
"autoload": {
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

Eloquent

Model Factories

Likelihood Of Impact: High

Laravel's model factories feature has been totally rewritten to support classes and is not compatible with Laravel 7.x style factories. However, to ease the upgrade process, a new `laravel/legacy-factories` package has been created to continue using your existing factories with Laravel 8.x. You may install this package via Composer:

```
composer require laravel/legacy-factories
```

The `Castable` Interface

Likelihood Of Impact: Low

The `castUsing` method of the `Castable` interface has been updated to accept an array of arguments. If you are implementing this interface you should update your implementation accordingly:

```
public static function castUsing(array $arguments);
```

Increment / Decrement Events

Likelihood Of Impact: Low

Proper "update" and "save" related model events will now be dispatched when executing the `increment` or `decrement` methods on Eloquent model instances.

Events

The `Dispatcher` Contract

Likelihood Of Impact: Low

The `listen` method of the `Illuminate\Contracts\Events\Dispatcher` contract has been updated to make the `$listener` property optional. This change was made to support automatic detection of handled event types via reflection. If you are manually implementing this interface, you should update your implementation accordingly:

```
public function listen($events, $listener = null);
```

Framework

Maintenance Mode Updates

Likelihood Of Impact: Optional

The maintenance mode feature of Laravel has been improved in Laravel 8.x. Pre-rendering the maintenance mode template is now supported and eliminates the chances of end users encountering errors during maintenance mode. However, to support this, the following lines must be added to your `public/index.php` file. These lines should be placed directly under the existing `LARAVEL_START` constant definition:

```
define('LARAVEL_START', microtime(true));

if (file_exists(__DIR__.'/../storage/framework/maintenance.php')) {
    require __DIR__.'/../storage/framework/maintenance.php';
}
```

The `php artisan down --message` Option

Likelihood Of Impact: Medium

The `--message` option of the `php artisan down` command has been removed. As an alternative, consider pre-

rendering your maintenance mode views with the message of your choice.

The `php artisan serve --no-reload` Option

Likelihood Of Impact: Low

A `--no-reload` option has been added to the `php artisan serve` command. This will instruct the built-in server to not reload the server when environment file changes are detected. This option is primarily helpful when running Laravel Dusk tests in a CI environment.

Manager `$app` Property

Likelihood Of Impact: Low

The previously deprecated `$app` property of the `Illuminate\Support\Manager` class has been removed. If you were relying on this property, you should use the `$container` property instead.

The `elixir` Helper

Likelihood Of Impact: Low

The previously deprecated `elixir` helper has been removed. Applications still using this method are encouraged to upgrade to Laravel Mix.

Mail

The `sendNow` Method

Likelihood Of Impact: Low

The previously deprecated `sendNow` method has been removed. Instead, please use the `send` method.

Pagination

Pagination Defaults

Likelihood Of Impact: High

The paginator now uses the Tailwind CSS framework for its default styling. In order to keep using Bootstrap, you should add the following method call to the `boot` method of your application's `AppServiceProvider`:

```
use Illuminate\Pagination\Paginator;

Paginator::useBootstrap();
```

Queue

The `retryAfter` Method

Likelihood Of Impact: High

For consistency with other features of Laravel, the `retryAfter` method and `retryAfter` property of queued jobs, mailers, notifications, and listeners has been renamed to `backoff`. You should update the name of this method / property in the relevant classes in your application.

The `timeoutAt` Property

Likelihood Of Impact: High

The `timeoutAt` property of queued jobs, notifications, and listeners has been renamed to `retryUntil`. You should

update the name of this property in the relevant classes in your application.

The `allOnQueue()` / `allOnConnection()` Methods

Likelihood Of Impact: High

For consistency with other dispatching methods, the `allOnQueue()` and `allOnConnection()` methods used with job chaining have been removed. You may use the `onQueue()` and `onConnection()` methods instead. These methods should be called before calling the `dispatch` method:

```
ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

Note that this change only affects code using the `withChain` method. The `allOnQueue()` and `allOnConnection()` are still available when using the global `dispatch()` helper.

Failed Jobs Table Batch Support

Likelihood Of Impact: Optional

If you plan to use the job batching features of Laravel 8.x, your `failed_jobs` database table will need to be updated. First, a new `uuid` column should be added to your table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('failed_jobs', function (Blueprint $table) {
    $table->string('uuid')->after('id')->nullable()->unique();
});
```

Next, the `failed.driver` configuration option within your `queue` configuration file should be updated to `database-uuids`.

In addition, you may wish to generate UUIDs for your existing failed jobs:

```
DB::table('failed_jobs')->whereNull('uuid')->cursor()->each(function ($job) {
    DB::table('failed_jobs')
        ->where('id', $job->id)
        ->update(['uuid' => (string) Illuminate\Support\Str::uuid()]);
});
```

Routing

Automatic Controller Namespace Prefixing

Likelihood Of Impact: Optional

In previous releases of Laravel, the `RouteServiceProvider` class contained a `$namespace` property with a value of `App\Http\Controllers`. This value of this property was used to automatically prefix controller route declarations and controller route URL generation such as when calling the `action` helper.

In Laravel 8, this property is set to `null` by default. This allows your controller route declarations to use the standard PHP callable syntax, which provides better support for jumping to the controller class in many IDEs:

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

// Using PHP callable syntax...
Route::get('/users', [UserController::class, 'index']);

// Using string syntax...
Route::get('/users', 'App\Http\Controllers\UserController@index');
```

In most cases this won't impact applications that are being upgraded because your `RouteServiceProvider` will still contain the `$namespace` property with its previous value. However, if you upgrade your application by creating a brand new Laravel project, you may encounter this as a breaking change.

If you would like to continue using the original auto-prefixed controller routing, you can simply set the value of the `$namespace` property within your `RouteServiceProvider` and update the route registrations within the `boot` method to use the `$namespace` property:

```
class RouteServiceProvider extends ServiceProvider
{
    /**
     * The path to the "home" route for your application.
     *
     * This is used by Laravel authentication to redirect users after login.
     *
     * @var string
     */
    public const HOME = '/home';

    /**
     * If specified, this namespace is automatically applied to your controller routes.
     *
     * In addition, it is set as the URL generator's root namespace.
     *
     * @var string
     */
    protected $namespace = 'App\Http\Controllers';

    /**
     * Define your route model bindings, pattern filters, etc.
     *
     * @return void
     */
    public function boot()
    {
        $this->configureRateLimiting();

        $this->routes(function () {
            Route::middleware('web')
                ->namespace($this->namespace)
                ->group(base_path('routes/web.php'));

            Route::prefix('api')
                ->middleware('api')
                ->namespace($this->namespace)
                ->group(base_path('routes/api.php'));
        });
    }

    /**
     * Configure the rate limiters for the application.
     *
     * @return void
     */
    protected function configureRateLimiting()
    {
        RateLimiter::for('api', function (Request $request) {
            return Limit::perMinute(60)->by(optional($request->user())->id ?: $request->ip());
        });
    }
}
```

Scheduling

The `cron-expression` Library

Likelihood Of Impact: Low

Laravel's dependency on `dragonmantank/cron-expression` has been updated from `2.x` to `3.x`. This should not cause any breaking change in your application unless you are interacting with the `cron-expression` library directly. If you are interacting with this library directly, please review its change log.

Session

The `Session` Contract

Likelihood Of Impact: Low

The `Illuminate\Contracts\Session\Session` contract has received a new `pull` method. If you are implementing this contract manually, you should update your implementation accordingly:

```
/**
 * Get the value of a given key and then forget it.
 *
 * @param  string  $key
 * @param  mixed  $default
 * @return mixed
 */
public function pull($key, $default = null);
```

Testing

The `decodeResponseJson` Method

Likelihood Of Impact: Low

The `decodeResponseJson` method that belongs to the `Illuminate\Testing\TestResponse` class no longer accepts any arguments. Please consider using the `json` method instead.

The `assertExactJson` Method

Likelihood Of Impact: Medium

The `assertExactJson` method now requires numeric keys of compared arrays to match and be in the same order. If you would like to compare JSON against an array without requiring numerically keyed arrays to have the same order, you may use the `assertSimilarJson` method instead.

Validation

Database Rule Connections

Likelihood Of Impact: Low

The `unique` and `exists` rules will now respect the specified connection name (accessed via the model's `getConnectionName` method) of Eloquent models when performing queries.

Miscellaneous

We also encourage you to view the changes in the `laravel/laravel` GitHub repository. While many of these changes are not required, you may wish to keep these files in sync with your application. Some of these changes will be covered in this upgrade guide, but others, such as changes to configuration files or comments, will not be. You can easily view the changes with the GitHub comparison tool and choose which updates are important to you.

Prologue

Contribution Guide

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

Bug Reports

To encourage active collaboration, Laravel strongly encourages pull requests, not just bug reports. "Bug reports" may also be sent in the form of a pull request containing a failing test.

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.

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 Framework
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Horizon
- Laravel Jetstream
- Laravel Passport
- Laravel Sanctum
- Laravel Scout
- Laravel Socialite
- Laravel Telescope
- Laravel Website

Support Questions

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

- GitHub Discussions
- Laracasts Forums

- Laravel.io Forums
- StackOverflow
- Discord
- Larachat
- IRC

Core Development Discussion

You may propose new features or improvements of existing Laravel behavior in the Laravel Ideas issue 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 stable branch or to the current LTS branch. 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.

Major new features should always be sent to the `master` branch, which contains the upcoming release.

If you are unsure if your feature qualifies as a major or minor, please ask Taylor Otwell in the `#internals` channel of the Laravel Discord server.

Compiled Assets

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

Security Vulnerabilities

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

Coding Style

Laravel follows the PSR-2 coding standard and the PSR-4 autoloading standard.

PHPDoc

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

```
/**
 * Register a binding with the container.
 *
 * @param  string|array  $abstract
 * @param  \Closure|string|null  $concrete
 * @param  bool  $shared
 * @return void
 *
```

```
 * @throws \Exception
 */
public function bind($abstract, $concrete = null, $shared = false)
{
    //
}
```

StyleCI

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

Code of Conduct

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

- Participants will be tolerant of opposing views.
- Participants must ensure that their language and actions are free of personal attacks and disparaging personal remarks.
- When interpreting the words and actions of others, participants should always assume good intentions.
- Behavior that can be reasonably considered harassment will not be tolerated.

Getting Started

Installation

- Meet Laravel
  - Why Laravel?
- Your First Laravel Project
  - Getting Started On macOS
  - Getting Started On Windows
  - Getting Started On Linux
  - Installation Via Composer
- Initial Configuration
- Next Steps
  - Laravel The Full Stack Framework
  - Laravel The API Backend

Meet Laravel

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

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

Whether you are new to PHP or 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 Vapor allow you to run your Laravel application at nearly limitless scale on AWS's latest serverless technology.

A Community Framework

Laravel combines the best packages in the PHP ecosystem to offer the most robust and developer friendly framework available. In addition, thousands of talented developers from around the world have contributed to

the framework. Who knows, maybe you'll even become a Laravel contributor.

Your First Laravel Project

We want it to be as easy as possible to get started with Laravel. There are a variety of options for developing and running a Laravel project on your own computer. While you may wish to explore these options at a later time, Laravel provides Sail, a built-in solution for running your Laravel project using Docker.

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

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

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

Getting Started On macOS

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

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

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

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

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail `up` command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

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

> TIP To continue learning more about Laravel Sail, review its complete documentation.

Getting Started On Windows

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

> TIP After installing and enabling WSL2, you should ensure that Docker Desktop is configured to use the WSL2 backend.

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

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

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

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

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail `up` command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

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

> TIP To continue learning more about Laravel Sail, review its complete documentation.

Developing Within WSL2

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

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

Getting Started On Linux

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

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

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

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

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail `up` command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

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

> TIP To continue learning more about Laravel Sail, review its complete documentation.

Installation Via Composer

If your computer already has PHP and Composer installed, you may create a new Laravel project by using Composer directly. After the application has been created, you may start Laravel's local development server using the Artisan CLI's `serve` command:

```
composer create-project laravel/laravel example-app

cd example-app

php artisan serve
```

The Laravel Installer

Or, you may install the Laravel Installer as a global Composer dependency:

```
composer global require laravel/installer
```

```
laravel new example-app
```

```
php artisan serve
```

Make sure to place Composer's system-wide vendor bin directory in your `$PATH` so the `laravel` executable can be located by your system. This directory exists in different locations based on your operating system; however, some common locations include:

- macOS: `$HOME/.composer/vendor/bin`
- Windows: `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin`
- GNU / Linux Distributions: `$HOME/.config/composer/vendor/bin` or `$HOME/.composer/vendor/bin`

Initial Configuration

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

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

Environment Based Configuration

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

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

> TIP For more information about the `.env` file and environment based configuration, check out the full configuration documentation.

Next Steps

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

- Request Lifecycle
- Configuration
- Directory Structure
- 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 using a single-page application hybrid technology like Inertia.js. This is the most common way to use the Laravel framework.

If this is how you plan to use Laravel, you may want to check out our documentation on routing, views, or the Eloquent ORM. In addition, you might be interested in learning about community packages like Livewire and Inertia.js. 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 Laravel Mix.

> TIP 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.

Getting Started

Configuration

- Introduction
- Environment Configuration
  - Environment Variable Types
  - Retrieving Environment Configuration
  - Determining The Current Environment
- Accessing Configuration Values
- Configuration Caching
- Debug Mode
- Maintenance Mode

Introduction

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

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

Environment Configuration

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

To make this a cinch, Laravel utilizes the DotEnv PHP library. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to `.env`.

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

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

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

Environment File Security

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

Environment Variable Types

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

`.env` Value	`env()` Value
true	(bool) true

(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) ''
(empty)	(string) ''
null	(null) null
(null)	(null) null

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

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

Retrieving Environment Configuration

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

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

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

Determining The Current Environment

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

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

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

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

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

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

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

Accessing Configuration Values

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

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

// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

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

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

Configuration Caching

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

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

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

Debug Mode

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

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

Maintenance Mode

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

To enable maintenance mode, execute the `down` Artisan command:

```
php artisan down
```

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

```
php artisan down --retry=60
```

Bypassing Maintenance Mode

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

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

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

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

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

Pre-Rendering The Maintenance Mode View

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

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

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

Redirecting Maintenance Mode Requests

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

```
php artisan down --redirect=/
```

Disabling Maintenance Mode

To disable maintenance mode, use the `up` command:

```
php artisan up
```

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

Maintenance Mode & Queues

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

Alternatives To Maintenance Mode

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

Getting Started

Directory Structure

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

Introduction

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

The Root Directory

The App Directory

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

The Bootstrap Directory

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

The Config Directory

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

The Database Directory

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

The Public Directory

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

The Resources Directory

The `resources` directory contains your views as well as your raw, un-compiled assets such as CSS or JavaScript. This directory also houses all of your language files.

The Routes Directory

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

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

The `api.php` file contains routes that the `RouteServiceProvider` places in the `api` middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.

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

The `channels.php` file is where you may register all of the event broadcasting channels that your application supports.

The Storage Directory

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

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

The Tests Directory

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

The Vendor Directory

The `vendor` directory contains your Composer dependencies.

The App Directory

The majority of your application is housed in the `app` directory. By default, this directory is namespaced under `App` and is autoloaded by Composer using the PSR-4 autoloading standard.

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

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

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

The Broadcasting Directory

The `Broadcasting` directory contains all of the broadcast channel classes for your application. These classes are generated using the `make:channel` command. This directory does not exist by default, but will be created for you when you create your first channel. To learn more about channels, check out the documentation on event broadcasting.

The Console Directory

The `Console` directory contains all of the custom Artisan commands for your application. These commands may be generated using the `make:command` command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your scheduled tasks are defined.

The Events Directory

This directory does not exist by default, but will be created for you by the `event:generate` and `make:event` Artisan commands. The `Events` directory houses event classes. Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

The Exceptions Directory

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

The Http Directory

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

The Jobs Directory

This directory does not exist by default, but will be created for you if you execute the `make:job` Artisan command. The `Jobs` directory houses the queueable jobs for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are sometimes referred to as "commands" since they are an implementation of the command pattern.

The Listeners Directory

This directory does not exist by default, but will be created for you if you execute the `event:generate` or `make:listener` Artisan commands. The `Listeners` directory contains the classes that handle your events. Event listeners receive an event instance and perform logic in response to the event being fired. For example, a

`UserRegistered` event might be handled by a `SendWelcomeEmail` listener.

The Mail Directory

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

The Models Directory

The `Models` directory contains all of your Eloquent model classes. The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

The Notifications Directory

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

The Policies Directory

This directory does not exist by default, but will be created for you if you execute the `make:policy` Artisan command. The `Policies` directory contains the authorization policy classes for your application. Policies are used to determine if a user can perform a given action against a resource.

The Providers Directory

The `Providers` directory contains all of the service providers for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.

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

The Rules Directory

This directory does not exist by default, but will be created for you if you execute the `make:rule` Artisan command. The `Rules` directory contains the custom validation rule objects for your application. Rules are used to encapsulate complicated validation logic in a simple object. For more information, check out the validation documentation.

Getting Started

Starter Kits

- Introduction
- Laravel Breeze
  - Installation
- Laravel Jetstream

Introduction

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

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

Laravel Breeze

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is made up of simple Blade templates styled with Tailwind CSS. Breeze provides a wonderful starting point for beginning a fresh Laravel application.

Installation

First, you should create a new Laravel application, configure your database, and run your database migrations:

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

cd example-app

php artisan migrate
```

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

```
composer require laravel/breeze --dev
```

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

```
php artisan breeze:install

npm install

npm run dev
```

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

> TIP To learn more about compiling your application's CSS and JavaScript, check out the Laravel Mix documentation.

Laravel Jetstream

While Laravel Breeze provides a simple and minimal starting point for building a Laravel application,

Jetstream augments that functionality with more robust features and additional frontend technology stacks. **For those brand new to Laravel, we recommend learning the ropes with Laravel Breeze before graduating to Laravel Jetstream.**

Jetstream provides a beautifully designed application scaffolding for Laravel and includes login, registration, email verification, two-factor authentication, session management, API support via Laravel Sanctum, and optional team management. Jetstream is designed using Tailwind CSS and offers your choice of Livewire or Inertia.js driven frontend scaffolding.

Complete documentation for installing Laravel Jetstream can be found within the official Jetstream documentation.

Getting Started

Deployment

- Introduction
- Server Requirements
- Server Configuration
  - Nginx
- Optimization
  - Autoloader Optimization
  - Optimizing Configuration Loading
  - Optimizing Route Loading
  - Optimizing View Loading
- Debug Mode
- Deploying With Forge / Vapor

Introduction

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

Server Requirements

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

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

Server Configuration

Nginx

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

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

```
server {
    listen 80;
    server_name example.com;
    root /srv/example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-XSS-Protection "1; mode=block";
    add_header X-Content-Type-Options "nosniff";
```

```
    index index.php;

    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php/php7.4-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
    }

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

Optimization

Autoloader Optimization

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

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

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

Optimizing Configuration Loading

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

```
php artisan config:cache
```

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

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

Optimizing Route Loading

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

```
php artisan route:cache
```

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

Optimizing View Loading

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

```
php artisan view:cache
```

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

Debug Mode

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

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

Deploying With Forge / Vapor

Laravel Forge

If you aren't quite ready to manage your own server configuration or aren't comfortable configuring all of the various services needed to run a robust Laravel application, Laravel Forge is a wonderful alternative.

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

Laravel Vapor

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

Architecture Concepts

Request Lifecycle

- Introduction
- Lifecycle Overview
  - First Steps
  - HTTP / Console Kernels
  - Service Providers
  - Routing
  - Finishing Up
- Focus On Service Providers

Introduction

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

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

Lifecycle Overview

First Steps

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

The `index.php` file loads the Composer generated autoloader definition, and then retrieves an instance of the Laravel application from `bootstrap/app.php`. The first action taken by Laravel itself is to create an instance of the application / service container.

HTTP / Console Kernels

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

The HTTP kernel extends the `Illuminate\Foundation\Http\Kernel` class, which defines an array of `bootstrappers` that will be run before the request is executed. These bootstrappers configure error handling, configure logging, detect the application environment, and perform other tasks that need to be done before the request is actually handled. Typically, these classes handle internal Laravel configuration that you do not need to worry about.

The HTTP kernel also defines a list of HTTP middleware that all requests must pass through before being handled by the application. These middleware handle reading and writing the HTTP session, determining if the application is in maintenance mode, verifying the CSRF token, and more. We'll talk more about these soon.

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

Service Providers

One of the most important kernel bootstrapping actions is loading the service providers for your application. All of the service providers for the application are configured in the `config/app.php` configuration file's `providers` array.

Laravel will iterate through this list of providers and instantiate each of them. After instantiating the providers, the `register` method will be called on all of the providers. Then, once all of the providers have been registered, the `boot` method will be called on each provider.

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

You may be wondering why the `register` method of every service provider is called before calling the `boot` method on any service providers. The answer is simple. By calling the `register` method of every service provider first, service providers may depend on every container binding being registered and available by the time the `boot` method is executed.

Routing

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

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

Middleware provide a convenient mechanism for filtering or examining HTTP requests entering your application. For example, Laravel includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application. Some middleware are assigned to all routes within the application, like those defined in the `$middleware` property of your HTTP kernel, while some are only assigned to specific routes or route groups. You can learn more about middleware by reading the complete middleware documentation.

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

Finishing Up

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

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

Focus On Service Providers

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

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

By default, the `AppServiceProvider` is fairly empty. This provider is a great place to add your application's own

bootstrapping and service container bindings. For large applications, you may wish to create several service providers, each with more granular bootstrapping for specific services used by your application.

Architecture Concepts

Service Container

- Introduction
  - Zero Configuration Resolution
  - When To Use The Container
- Binding
  - Binding Basics
  - Binding Interfaces To Implementations
  - Contextual Binding
  - Binding Primitives
  - Binding Typed Variadics
  - Tagging
  - Extending Bindings
- Resolving
  - The Make Method
  - Automatic Injection
- Container Events
- PSR-11

Introduction

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

Let's look at a simple example:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;
use App\Models\User;

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

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

    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        $user = $this->users->find($id);

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

```
}
```

In this example, the `UserController` needs to retrieve users from a data source. So, we will **inject** a service that is able to retrieve users. In this context, our `UserRepository` most likely uses Eloquent to retrieve user information from the database. However, since the repository is injected, we are able to easily swap it out with another implementation. We are also able to easily "mock", or create a dummy implementation of the `UserRepository` when testing our application.

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

Zero Configuration Resolution

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

```
<?php

class Service
{
    //
}

Route::get('/', function (Service $service) {
    die(get_class($service));
});
```

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

Thankfully, many of the classes you will be writing when building a Laravel application automatically receive their dependencies via the container, including controllers, event listeners, middleware, and more. Additionally, you may type-hint dependencies in the `handle` method of queued jobs. Once you taste the power of automatic and zero configuration dependency injection it feels impossible to develop without it.

When To Use The Container

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

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

In many cases, thanks to automatic dependency injection and facades, you can build Laravel applications without **ever** manually binding or resolving anything from the container. **So, when would you ever manually interact with the container?** Let's examine two situations.

First, if you write a class that implements an interface and you wish to type-hint that interface on a route or class constructor, you must tell the container how to resolve that interface. Secondly, if you are writing a Laravel package that you plan to share with other Laravel developers, you may need to bind your package's services into the container.

Binding

Binding Basics

Simple Bindings

Almost all of your service container bindings will be registered within service providers, so most of these examples will demonstrate using the container in that context.

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

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

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

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

As mentioned, you will typically be interacting with the container within service providers; however, if you would like to interact with the container outside of a service provider, you may do so via the `App` facade:

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

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

> TIP There is no need to bind classes into the container if they do not depend on any interfaces. The container does not need to be instructed on how to build these objects, since it can automatically resolve these objects using reflection.

Binding A Singleton

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

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

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

Binding Instances

You may also bind an existing object instance into the container using the `instance` method. The given instance will always be returned on subsequent calls into the container:

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

$service = new Transistor(new PodcastParser);

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

Binding Interfaces To Implementations

A very powerful feature of the service container is its ability to bind an interface to a given implementation. For example, let's assume we have an `EventPusher` interface and a `RedisEventPusher` implementation. Once we have coded our `RedisEventPusher` implementation of this interface, we can register it with the service container like so:

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

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

This statement tells the container that it should inject the `RedisEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in the constructor of a class that is resolved by the container. Remember, controllers, event listeners, middleware, and various other types of classes within Laravel applications are always resolved using the container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
 * @param  \App\Contracts\EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
    $this->pusher = $pusher;
}
```

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, two controllers may depend on different implementations of the `Illuminate\Contracts\Filesystem\Filesystem` contract. Laravel provides a simple, fluent interface for defining this behavior:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('local');
          });

$this->app->when([VideoController::class, UploadController::class])
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('s3');
          });
```

Binding Primitives

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

```
$this->app->when('App\Http\Controllers\UserController')
          ->needs('$variableName')
          ->give($value);
```

Sometimes a class may depend on an array of tagged instances. Using the `giveTagged` method, you may easily inject all of the container bindings with that tag:

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

Binding Typed Variadics

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

```
<?php

use App\Models\Filter;
use App\Services\Logger;

class Firewall
{
```

```
    /**
     * The logger instance.
     *
     * @var \App\Services\Logger
     */
    protected $logger;

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

    /**
     * Create a new class instance.
     *
     * @param  \App\Services\Logger  $logger
     * @param  array  $filters
     * @return void
     */
    public function __construct(Logger $logger, Filter ...$filters)
    {
        $this->logger = $logger;
        $this->filters = $filters;
    }
}
```

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

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

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

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

Variadic Tag Dependencies

Sometimes a class may have a variadic dependency that is type-hinted as a given class (`Report ...$reports`). Using the `needs` and `giveTagged` methods, you may easily inject all of the container bindings with that tag for the given dependency:

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

Tagging

Occasionally, you may need to resolve all of a certain "category" of binding. For example, perhaps you are building a report analyzer that receives an array of many different `Report` interface implementations. After registering the `Report` implementations, you can assign them a tag using the `tag` method:

```
$this->app->bind(CpuReport::class, function () {
    //
});

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

```
});

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

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

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

Extending Bindings

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

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

Resolving

The `make` Method

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

```
use App\Services\Transistor;

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

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

```
use App\Services\Transistor;

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

If you are outside of a service provider in a location of your code that does not have access to the `$app` variable, you may use the `App` facade to resolve a class instance from the container:

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

$api = App::make(Transistor::class);
```

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

```
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 *
 * @param  \Illuminate\Container\Container
 * @return void
 */
public function __construct(Container $container)
{
    $this->container = $container;
}
```

Automatic Injection

Alternatively, and importantly, you may type-hint the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, middleware, and more. Additionally, you may type-hint

dependencies in the `handle` method of queued jobs. In practice, this is how most of your objects should be resolved by the container.

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

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     *
     * @var \App\Repositories\UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the user with the given ID.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }
}
```

Container Events

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

```
use App\Services\Transistor;

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

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

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

PSR-11

Laravel's service container implements the PSR-11 interface. Therefore, you may type-hint the PSR-11 container interface to obtain an instance of the Laravel container:

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

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

```
    //
});
```

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

Architecture Concepts

Service Providers

- Introduction
- Writing Service Providers
  - The Register Method
  - The Boot Method
- Registering Providers
- Deferred Providers

Introduction

Service providers are the central place of all Laravel application bootstrapping. Your own application, as well as all of Laravel's core services are bootstrapped via service providers.

But, what do we mean by "bootstrapped"? In general, we mean **registering** things, including registering service container bindings, event listeners, middleware, and even routes. Service providers are the central place to configure your application.

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

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

> TIP If you would like to learn more about how Laravel handles requests and works internally, check out our documentation on the Laravel request lifecycle.

Writing Service Providers

All service providers extend the `Illuminate\Support\ServiceProvider` class. Most service providers contain a `register` and a `boot` method. Within the `register` method, you should **only bind things into the service container**. You should never attempt to register any event listeners, routes, or any other piece of functionality within the `register` method.

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

```
php artisan make:provider RiakServiceProvider
```

The Register Method

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

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

```
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
```

```
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `App\Services\Riak\Connection` in the service container. If you 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.
     *
```

```
     * @return void
     */
    public function boot()
    {
        View::composer('view', function () {
            //
        });
    }
}
```

Boot Method Dependency Injection

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

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

/**
 * Bootstrap any application services.
 *
 * @param  \Illuminate\Contracts\Routing\ResponseFactory
 * @return void
 */
public function boot(ResponseFactory $response)
{
    $response->macro('serialized', function ($value) {
        //
    });
}
```

Registering Providers

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

To register your provider, add it to the array:

```
'providers' => [
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
],
```

Deferred Providers

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\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
```

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

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

Architecture Concepts

Facades

- Introduction
- When To Use Facades
  - Facades Vs. Dependency Injection
  - Facades Vs. Helper Functions
- How Facades Work
- Real-Time Facades
- Facade Class Reference

Introduction

Throughout the Laravel documentation, you will see examples of code that interacts with Laravel's features via "facades". Facades provide a "static" interface to classes that are available in the application's service container. Laravel ships with many facades which provide access to almost all of Laravel's features.

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

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

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

Route::get('/cache', function () {
    return Cache::get('key');
});
```

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

Helper Functions

To compliment facades, Laravel offers a variety of global "helper functions" that make it even easier to interact with common Laravel features. Some of the common helper functions you may interact with are `view`, `response`, `url`, `config`, and more. Each helper function offered by Laravel is documented with their corresponding feature; however, a complete list is available within the dedicated helper documentation.

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

```
use Illuminate\Support\Facades\Response;

Route::get('/users', function () {
    return Response::json([
        // ...
    ]);
});

Route::get('/users', function () {
    return response()->json([
        // ...
    ]);
});
```

When To Use Facades

Facades have many benefits. They provide a terse, memorable syntax that allows you to use Laravel's features

without remembering long class names that must be injected or configured manually. Furthermore, because of their unique usage of PHP's dynamic methods, they are easy to test.

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

Facades Vs. Dependency Injection

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

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

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

Route::get('/cache', function () {
    return Cache::get('key');
});
```

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

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

/**
 * A basic functional test example.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
         ->with('key')
         ->andReturn('value');

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

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

Facades Vs. Helper Functions

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

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

return view('profile');
```

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

```
Route::get('/cache', function () {
    return cache('key');
});
```

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

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

/**
 * A basic functional test example.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
         ->with('key')
         ->andReturn('value');

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

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

How Facades Work

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

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

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;

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

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

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

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

```
class Cache extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }
}
```

Instead, the `Cache` facade extends the base `Facade` class and defines the method `getFacadeAccessor()`. This method's job is to return the name of a service container binding. When a user references any static method on the `Cache` facade, Laravel resolves the `cache` binding from the service container 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 were a facade. To illustrate how this can be used, let's first examine some code that does not use real-time facades. For example, let's assume our `Podcast` model has a `publish` method. However, in order to publish the podcast, we need to inject a `Publisher` instance:

```
<?php

namespace App\Models;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @param  Publisher  $publisher
     * @return void
     */
    public function publish(Publisher $publisher)
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
    }
}
```

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

```
<?php

namespace App\Models;

use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @return void
     */
    public function publish()
    {
        $this->update(['publishing' => now()]);

        Publisher::publish($this);
    }
}
```

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

```
<?php

namespace Tests\Feature;

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

class PodcastTest extends TestCase
{
    use RefreshDatabase;
```

```
    /**
     * A test example.
     *
     * @return void
     */
    public function test_podcast_can_be_published()
    {
        $podcast = Podcast::factory()->create();

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

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

Facade Class Reference

Below you will find every facade and its underlying class. This is a useful tool for quickly digging into the API documentation for a given facade root. The service container binding key is also included where applicable.

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	app
Artisan	Illuminate\Contracts\Console\Kernel	artisan
Auth	Illuminate\Auth\AuthManager	auth
Auth (Instance)	Illuminate\Contracts\Auth\Guard	auth.driver
Blade	Illuminate\View\Compilers\BladeCompiler	blade.compiler
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\CacheManager	cache
Cache (Instance)	Illuminate\Cache\Repository	cache.store
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	db.connection
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Http	Illuminate\Http\Client\Factory	
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\LogManager	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	auth.password
Password (Instance)	Illuminate\Auth\Passwords\PasswordBroker	auth.password.broker
Queue	Illuminate\Queue\QueueManager	queue
Queue (Instance)	Illuminate\Contracts\Queue\Queue	queue.connection
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	redirect
Redis	Illuminate\Redis\RedisManager	redis
Redis (Instance)	Illuminate\Redis\Connections\Connection	redis.connection
Request	Illuminate\Http\Request	request
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (Instance)	Illuminate\Http\Response	
Route	Illuminate\Routing\Router	router
Schema	Illuminate\Database\Schema\Builder	

Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	`session.store`
Storage	Illuminate\Filesystem\FilesystemManager	`filesystem`
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	`filesystem.disk`
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`
View (Instance)	Illuminate\View\View	

The Basics

Routing

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

Basic Routing

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

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

The Default Route Files

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

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

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

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

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

Available Router Methods

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

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

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

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

Route::any('/', function () {
    //
});
```

Dependency Injection

You may type-hint any dependencies required by your route in your route's callback signature. The declared dependencies will automatically be resolved and injected into the callback by the Laravel service container. For example, you may type-hint the `Illuminate\Http\Request` class to have the current HTTP request automatically injected into your route callback:

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

CSRF Protection

Remember, any HTML forms pointing to `POST`, `PUT`, `PATCH`, or `DELETE` routes that are defined in the `web` routes file should include a CSRF token field. Otherwise, the request will be rejected. You can read more about CSRF protection in the CSRF documentation:

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

Redirect Routes

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

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

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

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

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

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

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

View Routes

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

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

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

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

Route Parameters

Required Parameters

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

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

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

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

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

Parameters & Dependency Injection

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

```
use Illuminate\Http\Request;

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

Optional Parameters

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

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

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

Regular Expression Constraints

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

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

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

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

Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

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

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

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

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

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

Global Constraints

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

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');
}
```

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

```
Route::get('/user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});
```

Encoded Forward Slashes

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

```
Route::get('/search/{search}', function ($search) {
    return $search;
})->where('search', '.*');
```

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

Named Routes

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

```
Route::get('/user/profile', function () {
    //
})->name('profile');
```

You may also specify route names for controller actions:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

NOTE Route names should always be unique.

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via Laravel's `route` and `redirect` helper functions:

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

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

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

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

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

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

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

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

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

TIP Sometimes, you may wish to specify request-wide default values for URL parameters, such as the current locale. To accomplish this, you may use the `URL::defaults` method.

Inspecting The Current Route

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

```
/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure  $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

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

Route Groups

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

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and `where`

conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` method before defining the group. Middleware are executed in the order they are listed in the array:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

Subdomain Routing

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

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

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

Route Prefixes

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

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

Route Name Prefixes

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

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

Route Model Binding

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

Implicit Binding

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

```
use App\Models\User;

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

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

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

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

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

// Controller method definition...
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```

Customizing The Key

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

```
use App\Models\Post;

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

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

```
/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

Custom Keys & Scoping

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

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

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

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

Explicit Binding

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

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::model('user', User::class);

    // ...
}
```

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

```
use App\Models\User;

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

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

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

Customizing The Resolution Logic

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

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::bind('user', function ($value) {
        return User::where('name', $value)->firstOrFail();
    });

    // ...
}
```

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

```
/**
 * Retrieve the model for a bound value.
 *
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
```

```
 */
public function resolveRouteBinding($value, $field = null)
{
    return $this->where('name', $value)->firstOrFail();
}
```

If a route is utilizing implicit binding scoping, the `resolveChildRouteBinding` method will be used to resolve the child binding of the parent model:

```
/**
 * Retrieve the child model for a bound value.
 *
 * @param  string  $childType
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveChildRouteBinding($childType, $value, $field)
{
    return parent::resolveChildRouteBinding($childType, $value, $field);
}
```

Fallback Routes

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

```
Route::fallback(function () {
    //
});
```

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

Rate Limiting

Defining Rate Limiters

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

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

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

/**
 * Configure the rate limiters for the application.
 *
 * @return void
 */
protected function configureRateLimiting()
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });
}
```

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

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function () {
        return response('Custom response...', 429);
    });
});
```

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

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100);
});
```

Segmenting Rate Limits

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

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100)->by($request->ip());
});
```

Multiple Rate Limits

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

```
RateLimiter::for('login', function (Request $request) {
    return [
        Limit::perMinute(500),
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

Attaching Rate Limiters To Routes

Rate limiters may be attached to routes or route groups using the `throttle` middleware. The throttle middleware accepts the name of the rate limiter you wish to assign to the route:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        //
    });

    Route::post('/video', function () {
        //
    });
});
```

Throttling With Redis

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

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

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are

called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

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

For convenience, you may use the `@method` Blade directive to generate the `_method` input field:

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

Accessing The Current Route

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

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

You may refer to the API documentation for both the underlying class of the Route facade and Route instance to review all of the methods that are available on the router and route classes.

Cross-Origin Resource Sharing (CORS)

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

> TIP For more information on CORS and CORS headers, please consult the MDN web documentation on CORS.

Route Caching

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

```
php artisan route:cache
```

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

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

```
php artisan route:clear
```

The Basics

Middleware

- Introduction
- Defining Middleware
- Registering Middleware
  - Global Middleware
  - Assigning Middleware To Routes
  - Middleware Groups
  - Sorting Middleware
- Middleware Parameters
- Terminable Middleware

Introduction

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

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

Defining Middleware

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

```
php artisan make:middleware EnsureTokenIsValid
```

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

```
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

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

As you can see, if the given `token` does not match our secret token, the middleware will return an HTTP redirect to the client; otherwise, the request will be passed further into the application. To pass the request deeper into the application (allowing the middleware to "pass"), you should call the `$next` callback with the `$request`.

It's best to envision middleware as a series of "layers" HTTP requests must pass through before they hit your application. Each layer can examine the request and even reject it entirely.

> TIP All middleware are resolved via the service container, so you may type-hint any dependencies you need within a middleware's constructor.

Middleware & Responses

Of course, a middleware can perform tasks before or after passing the request deeper into the application. For example, the following middleware would perform some task **before** the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // Perform action

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

However, this middleware would perform its task **after** the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        // Perform action

        return $response;
    }
}
```

Registering Middleware

Global Middleware

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

Assigning Middleware To Routes

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

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

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
```

```
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

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

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

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

```
Route::get('/', function () {
    //
})->middleware(['first', 'second']);
```

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

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

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

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

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

Route::middleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/', function () {
        //
    });

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

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

Middleware Groups

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

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

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:api',
```

```
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

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

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

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

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

Sorting Middleware

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

```
/**
 * The priority-sorted list of middleware.
 *
 * This forces non-global middleware to always be in the given order.
 *
 * @var array
 */
protected $middlewarePriority = [
    \Illuminate\Cookie\Middleware\EncryptCookies::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class,
    \Illuminate\Session\Middleware\AuthenticateSession::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    \Illuminate\Auth\Middleware\Authorize::class,
];
```

Middleware Parameters

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

}
```

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

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

Terminable Middleware

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

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

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

    /**
     * Handle tasks after the response has been sent to the browser.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Illuminate\Http\Response  $response
     * @return void
     */
    public function terminate($request, $response)
    {
        // ...
    }
}
```

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

When calling the `terminate` method on your middleware, Laravel will resolve a fresh instance of the middleware from the service container. If you would like to use the same middleware instance when the `handle` and `terminate` methods are called, register the middleware with the container using the container's `singleton` method. Typically this should be done in the `register` method of your `AppServiceProvider`:

```
use App\Http\Middleware\TerminatingMiddleware;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->app->singleton(TerminatingMiddleware::class);
}
```

The Basics

CSRF Protection

- Introduction
- Preventing CSRF Requests
  - Excluding URIs
- X-CSRF-Token
- X-XSRF-Token

Introduction

Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user. Thankfully, Laravel makes it easy to protect your application from cross-site request forgery (CSRF) attacks.

An Explanation Of The Vulnerability

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

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

```
<form action="https://your-application.com/user/email" method="POST">
    <input type="email" value="malicious-email@example.com">
</form>

<script>
    document.forms[0].submit();
</script>
```

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

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

Preventing CSRF Requests

Laravel automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the person actually making the requests to the application. Since this token is stored in the user's session and changes each time the session is regenerated, a malicious application is unable to access it.

The current session's CSRF token can be accessed via the request's session or via the `csrf_token` helper function:

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

Anytime you define an HTML form in your application, you should include a hidden CSRF `_token` field in the form so that the CSRF protection middleware can validate the request. For convenience, you may use the `@csrf`

Blade directive to generate the hidden token input field:

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

    <!-- Equivalent to... -->
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
</form>
```

The `App\Http\Middleware\VerifyCsrfToken` middleware, which is included in the `web` middleware group by default, will automatically verify that the token in the request input matches the token stored in the session. When these two tokens match, we know that the authenticated user is the one initiating the request.

CSRF Tokens & SPAs

If you are building 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 the `App\Providers\RouteServiceProvider` applies to all routes in the `routes/web.php` file. However, you may also exclude the routes by adding their URIs to the `$except` property of the `VerifyCsrfToken` middleware:

```
<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ];
}
```

TIP For convenience, the CSRF middleware is automatically disabled for all routes when running tests.

X-CSRF-TOKEN

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

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

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

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

X-XSRF-TOKEN

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

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

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

The Basics

Controllers

- Introduction
- Writing Controllers
  - Basic Controllers
  - Single Action Controllers
- Controller Middleware
- Resource Controllers
  - Partial Resource Routes
  - Nested Resources
  - Naming Resource Routes
  - Naming Resource Route Parameters
  - Scoping Resource Routes
  - Localizing Resource URIs
  - Supplementing Resource Controllers
- Dependency Injection & 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

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

```
<?php

namespace App\Http\Controllers;

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

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

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

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

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

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

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

Single Action Controllers

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

```
<?php

namespace App\Http\Controllers;

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

class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function __invoke()
    {
        // ...
    }
}
```

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

```
use App\Http\Controllers\ProvisionServer;

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

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

```
php artisan make:controller ProvisionServer --invokable
```

TIP Controller stubs may be customized using stub publishing

Controller Middleware

Middleware may be assigned to the controller's routes in your route files:

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

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

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

Controllers also allow you to register middleware using a closure. This provides a convenient way to define an inline middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
```

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

Resource Controllers

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 by running the `route:list` Artisan command.

You may even register many resource controllers at once by passing an array to the `resources` method:

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

Actions Handled By Resource Controller

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

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 --resource --model=Photo
```

Partial Resource Routes

When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions:

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

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

```
Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

API Resource Routes

When declaring resource routes that will be consumed by APIs, you will commonly want to exclude routes that present HTML templates such as `create` and `edit`. For convenience, you may use the `apiResource` method to automatically exclude these two routes:

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

Route::apiResource('photos', PhotoController::class);
```

You may register many API resource controllers at once by passing an array to the `apiResources` method:

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

Route::apiResources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

To quickly generate an API resource controller that does not include the `create` or `edit` methods, use the `--api` switch when executing the `make:controller` command:

```
php artisan make:controller API/PhotoController --api
```

Nested Resources

Sometimes you may need to define routes to a nested resource. For example, a photo resource may have multiple comments that may be attached to the photo. To nest the resource controllers, you may use "dot" notation in your route declaration:

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

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

This route will register a nested resource that may be accessed with URIs like the following:

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

Scoping Nested Resources

Laravel's implicit model binding feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the `scoped` method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by. For more information on how to accomplish this, please see the documentation on scoping resource routes.

Shallow Nesting

Often, it is not entirely necessary to have both the parent and the child IDs within a URI since the child ID is already a unique identifier. When using unique identifiers such as auto-incrementing primary keys to identify your models in URI segments, you may choose to use "shallow nesting":

```
use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();
```

This route definition will define the following routes:

Verb	URI	Action	Route Name
GET	`/photos/{photo}/comments`	index	photos.comments.index
GET	`/photos/{photo}/comments/create`	create	photos.comments.create

POST	`/photos/{photo}/comments`	store	photos.comments.store
GET	`/comments/{comment}`	show	comments.show
GET	`/comments/{comment}/edit`	edit	comments.edit
PUT/PATCH	`/comments/{comment}`	update	comments.update
DELETE	`/comments/{comment}`	destroy	comments.destroy

Naming Resource Routes

By default, all resource controller actions have a route name; however, you can override these names by passing a `names` array with your desired route names:

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

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

Naming Resource Route Parameters

By default, `Route::resource` will create the route parameters for your resource routes based on the "singularized" version of the resource name. You can easily override this on a per resource basis using the `parameters` method. The array passed into the `parameters` method should be an associative array of resource names and parameter names:

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

The example above generates the following URI for the resource's `show` route:

```
/users/{admin_user}
```

Scoping Resource Routes

Laravel's scoped implicit model binding feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the `scoped` method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by:

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

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

This route will register a scoped nested resource that may be accessed with URIs like the following:

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

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

Localizing Resource URIs

By default, `Route::resource` will create resource URIs using English verbs. If you need to localize the `create` and `edit` action verbs, you may use the `Route::resourceVerbs` method. This may be done at the beginning of the `boot` method within your application's `App\Providers\RouteServiceProvider`:

```
/**
 * Define your route model bindings, pattern filters, etc.
```

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

    // ...
}
```

Once the verbs have been customized, a resource route registration such as `Route::resource('fotos', PhotoController::class)` will produce the following URIs:

```
/fotos/crear

/fotos/{foto}/editar
```

Supplementing Resource Controllers

If you need to add additional routes to a resource controller beyond the default set of resource routes, you should define those routes before your call to the `Route::resource` method; otherwise, the routes defined by the `resource` method may unintentionally take precedence over your supplemental routes:

```
use App\Http\Controller\PhotoController;

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

> TIP Remember to keep your controllers focused. If you find yourself routinely needing methods outside of the typical set of resource actions, consider splitting your controller into two, smaller controllers.

Dependency Injection & Controllers

Constructor Injection

The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The declared dependencies will automatically be resolved and injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

Method Injection

In addition to constructor injection, you may also type-hint dependencies on your controller's methods. A

common use-case for method injection is injecting the `Illuminate\Http\Request` instance into your controller methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

        //
    }
}
```

If your controller method is also expecting input from a route parameter, list your route arguments after your other dependencies. For example, if your route is defined like so:

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

Route::put('/user/{id}', [UserController::class, 'update']);
```

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

The Basics

HTTP Requests

- Introduction
- Interacting With The Request
  - Accessing The Request
  - Request Path & Method
  - Request Headers
  - Request IP Address
  - Content Negotiation
  - PSR-7 Requests
- Input
  - Retrieving Input
  - Determining If Input Is Present
  - Old Input
  - Cookies
  - Input Trimming & Normalization
- Files
  - Retrieving Uploaded Files
  - Storing Uploaded Files
- Configuring Trusted Proxies

Introduction

Laravel's `Illuminate\Http\Request` class provides an object-oriented way to interact with the current HTTP request being handled by your application as well retrieve the input, cookies, and files that were submitted with the request.

Interacting With The Request

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` class on your route closure or controller method. The incoming request instance will automatically be injected by the Laravel service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

As mentioned, you may also type-hint the `Illuminate\Http\Request` class on a route closure. The service container will automatically inject the incoming request into the closure when it is executed:

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

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

Dependency Injection & Route Parameters

If your controller method is also expecting input from a route parameter you should list your route parameters after your other dependencies. For example, if your route is defined like so:

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

Route::put('/user/{id}', [UserController::class, 'update']);
```

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

Request Path & Method

The `Illuminate\Http\Request` instance provides a variety of methods for examining the incoming HTTP request and extends the `Symfony\Component\HttpFoundation\Request` class. We will discuss a few of the most important methods below.

Retrieving The Request Path

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

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

Inspecting The Request Path / Route

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

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

Using the `routeIs` method, you may determine if the incoming request has matched a named route:

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

Retrieving The Request URL

To retrieve the full URL for the incoming request you may use the `url` or `fullUrl` methods. The `url` method will

return the URL without the query string, while the `fullUrl` method includes the query string:

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

$urlWithQueryString = $request->fullUrl();
```

Retrieving The Request Method

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

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

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

Request Headers

You may retrieve a request header from the `Illuminate\Http\Request` instance using the `header` method. If the header is not present on the request, `null` will be returned. However, the `header` method accepts an optional second argument that will be returned if the header is not present on the request:

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

The `hasHeader` method may be used to determine if the request contains a given header:

```
if ($request->hasHeader('X-Header-Name')) {
    //
}
```

For convenience, the `bearerToken` may be used to a bearer token from the `Authorization` header. If no such header is present, an empty string will be returned:

```
$token = $request->bearerToken();
```

Request IP Address

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

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

Content Negotiation

Laravel provides several methods for inspecting the incoming request's requested content types via the `Accept` header. First, the `getAcceptableContentTypes` method will return an array containing all of the content types accepted by the request:

```
$contentTypes = $request->getAcceptableContentTypes();
```

The `accepts` method accepts an array of content types and returns `true` if any of the content types are accepted by the request. Otherwise, `false` will be returned:

```
if ($request->accepts(['text/html', 'application/json'])) {
    // ...
}
```

You may use the `prefers` method to determine which content type out of a given array of content types is most preferred by the request. If none of the provided content types are accepted by the request, `null` will be returned:

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

Since many applications only serve HTML or JSON, you may use the `expectsJson` method to quickly determine

if the incoming request expects a JSON response:

```
if ($request->expectsJson()) {
    // ...
}
```

PSR-7 Requests

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

```
composer require symfony/psr-http-message-bridge
composer require nyholm/psr7
```

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

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

Route::get('/', function (ServerRequestInterface $request) {
    //
});
```

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

Input

Retrieving Input

Retrieving All Input Data

You may retrieve all of the incoming request's input data as an `array` using the `all` method. This method may be used regardless of whether the incoming request is from an HTML form or is an XHR request:

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

Retrieving An Input Value

Using a few simple methods, you may access all of the user input from your `Illuminate\Http\Request` instance without worrying about which HTTP verb was used for the request. Regardless of the HTTP verb, the `input` method may be used to retrieve user input:

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

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

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

When working with forms that contain array inputs, use "dot" notation to access the arrays:

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

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

You may call the `input` method without any arguments in order to retrieve all of the input values as an associative array:

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

Retrieving Input From The Query String

While the `input` method retrieves values from the entire request payload (including the query string), the `query` method will only retrieve values from the query string:

```
$name = $request->query('name');
```

If the requested query string value data is not present, the second argument to this method will be returned:

```
$name = $request->query('name', 'Helen');
```

You may call the `query` method without any arguments in order to retrieve all of the query string values as an associative array:

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

Retrieving JSON Input Values

When sending JSON requests to your application, you may access the JSON data via the `input` method as long as the `Content-Type` header of the request is properly set to `application/json`. You may even use "dot" syntax to retrieve values that are nested within JSON arrays:

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

Retrieving Boolean Input Values

When dealing with HTML elements like checkboxes, your application may receive "truthy" values that are actually strings. For example, "true" or "on". For convenience, you may use the `boolean` method to retrieve these values as booleans. The `boolean` method returns `true` for 1, "1", true, "true", "on", and "yes". All other values will return `false`:

```
$archived = $request->boolean('archived');
```

Retrieving Input Via Dynamic Properties

You may also access user input using dynamic properties on the `Illuminate\Http\Request` instance. For example, if one of your application's forms contains a `name` field, you may access the value of the field like so:

```
$name = $request->name;
```

When using dynamic properties, Laravel will first look for the parameter's value in the request payload. If it is not present, Laravel will search for the field in the matched route's parameters.

Retrieving A Portion Of The Input Data

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

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

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

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

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

> NOTE The `only` method returns all of the key / value pairs that you request; however, it will not return key / value pairs that are not present on the request.

Determining If Input Is Present

You may use the `has` method to determine if a value is present on the request. The `has` method returns `true` if the value is present on the request:

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

When given an array, the `has` method will determine if all of the specified values are present:

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

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

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

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

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

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

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

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

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

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

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

Old Input

Laravel allows you to keep input from one request during the next request. This feature is particularly useful for re-populating forms after detecting validation errors. However, if you are using Laravel's included validation features, it is possible that you will not need to manually use these session input flashing methods directly, as some of Laravel's built-in validation facilities will call them automatically.

Flashing Input To The Session

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

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

You may also use the `flashOnly` and `flashExcept` methods to flash a subset of the request data to the session. These methods are useful for keeping sensitive information such as passwords out of the session:

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

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

Flashing Input Then Redirecting

Since you often will want to flash input to the session and then redirect to the previous page, you may easily chain input flashing onto a redirect using the `withInput` method:

```
return redirect('form')->withInput();

return redirect()->route('user.create')->withInput();

return redirect('form')->withInput(
    $request->except('password')
```

```
);
```

Retrieving Old Input

To retrieve flashed input from the previous request, invoke the `old` method on an instance of `Illuminate\Http\Request`. The `old` method will pull the previously flashed input data from the session:

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

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper to repopulate the form. If no old input exists for the given field, `null` will be returned:

```
<input type="text" name="username" value="{{ old('username') }}">
```

Cookies

Retrieving Cookies From Requests

All cookies created by the Laravel framework are encrypted and signed with an authentication code, meaning they will be considered invalid if they have been changed by the client. To retrieve a cookie value from the request, use the `cookie` method on an `Illuminate\Http\Request` instance:

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

Input Trimming & Normalization

By default, Laravel includes the `App\Http\Middleware\TrimStrings` and `App\Http\Middleware\ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the global middleware stack by the `App\Http\Kernel` class. These middleware will automatically trim all incoming string fields on the request, as well as convert any empty string fields to `null`. This allows you to not have to worry about these normalization concerns in your routes and controllers.

If you would like to disable this behavior, you may remove the two middleware from your application's middleware stack by removing them from the `$middleware` property of your `App\Http\Kernel` class.

Files

Retrieving Uploaded Files

You may retrieve uploaded files from an `Illuminate\Http\Request` instance using the `file` method or using dynamic properties. The `file` method returns an instance of the `Illuminate\Http\UploadedFile` class, which extends the PHP `SplFileInfo` class and provides a variety of methods for interacting with the file:

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

$file = $request->photo;
```

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

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

Validating Successful Uploads

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

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

File Paths & Extensions

The `UploadedFile` class also contains methods for accessing the file's fully-qualified path and its extension. The `extension` method will attempt to guess the file's extension based on its contents. This extension may be different from the extension that was supplied by the client:

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

$extension = $request->photo->extension();
```

Other File Methods

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

Storing Uploaded Files

To store an uploaded file, you will typically use one of your configured filesystems. The `UploadedFile` class has a `store` method that will move an uploaded file to one of your disks, which may be a location on your local filesystem or a cloud storage location like Amazon S3.

The `store` method accepts the path where the file should be stored relative to the filesystem's configured root directory. This path should not contain a filename, since a unique ID will automatically be generated to serve as the filename.

The `store` method also accepts an optional second argument for the name of the disk that should be used to store the file. The method will return the path of the file relative to the disk's root:

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

If you do not want a filename to be automatically generated, you may use the `storeAs` method, which accepts the path, filename, and disk name as its arguments:

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

TIP For more information about file storage in Laravel, check out the complete file storage documentation.

Configuring Trusted Proxies

When running your applications behind a load balancer that terminates TLS / SSL certificates, you may notice your application sometimes does not generate HTTPS links when using the `url` helper. Typically this is because your application is being forwarded traffic from your load balancer on port 80 and does not know it should generate secure links.

To solve this, you may use the `App\Http\Middleware\TrustProxies` middleware that is included in your Laravel application, which allows you to quickly customize the load balancers or proxies that should be trusted by your application. Your trusted proxies should be listed as an array on the `$proxies` property of this middleware. In addition to configuring the trusted proxies, you may configure the proxy `$headers` that should be trusted:

```
<?php

namespace App\Http\Middleware;

use Fideloper\Proxy\TrustProxies as Middleware;
use Illuminate\Http\Request;

class TrustProxies extends Middleware
{
    /**
     * The trusted proxies for this application.
     *
     * @var string|array
```

```
     */
    protected $proxies = [
        '192.168.1.1',
        '192.168.1.2',
    ];

    /**
     * The headers that should be used to detect proxies.
     *
     * @var int
     */
    protected $headers = Request::HEADER_X_FORWARDED_ALL;
}
```

TIP If you are using AWS Elastic Load Balancing, your `$headers` value should be `Request::HEADER_X_FORWARDED_AWS_ELB`. For more information on the constants that may be used in the `$headers` property, check out Symfony's documentation on trusting proxies.

Trusting All Proxies

If you are using Amazon AWS or another "cloud" load balancer provider, you may not know the IP addresses of your actual balancers. In this case, you may use `*` to trust all proxies:

```
/**
 * The trusted proxies for this application.
 *
 * @var string|array
 */
protected $proxies = '*';
```

The Basics

HTTP Responses

- Creating Responses
  - Attaching Headers To Responses
  - Attaching Cookies To Responses
  - Cookies & Encryption
- Redirects
  - Redirecting To Named Routes
  - Redirecting To Controller Actions
  - Redirecting To External Domains
  - Redirecting With Flashed Session Data
- Other Response Types
  - View Responses
  - JSON Responses
  - File Downloads
  - File Responses
- Response Macros

Creating Responses

Strings & Arrays

All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:

```
Route::get('/', function () {
    return 'Hello World';
});
```

In addition to returning strings from your routes and controllers, you may also return arrays. The framework will automatically convert the array into a JSON response:

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

> TIP Did you know you can also return Eloquent collections from your routes or controllers? They will automatically be converted to JSON. Give it a shot!

Response Objects

Typically, you won't just be returning simple strings or arrays from your route actions. Instead, you will be returning full `Illuminate\Http\Response` instances or views.

Returning a full `Response` instance allows you to customize the response's HTTP status code and headers. A `Response` instance inherits from the `Symfony\Component\HttpFoundation\Response` class, which provides a variety of methods for building HTTP responses:

```
Route::get('/home', function () {
    return response('Hello World', 200)
                    ->header('Content-Type', 'text/plain');
});
```

Eloquent Models & Collections

You may also return Eloquent ORM models and collections directly from your routes and controllers. When you do, Laravel will automatically convert the models and collections to JSON responses while respecting the model's hidden attributes:

```
use App\Models\User;

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

Attaching Headers To Responses

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

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

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

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

Cache Control Middleware

Laravel includes a `cache.headers` middleware, which may be used to quickly set the `Cache-Control` header for a group of routes. If `etag` is specified in the list of directives, an MD5 hash of the response content will automatically be set as the ETag identifier:

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(function () {
    Route::get('/privacy', function () {
        // ...
    });

    Route::get('/terms', function () {
        // ...
    });
});
```

Attaching Cookies To Responses

You may attach a cookie to an outgoing `Illuminate\Http\Response` instance using the `cookie` method. You should pass the name, value, and number of minutes the cookie should be considered valid to this method:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

The `cookie` method also accepts a few more arguments which are used less frequently. Generally, these arguments have the same purpose and meaning as the arguments that would be given to PHP's native setcookie method:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

If you would like to ensure that a cookie is sent with the outgoing response but you do not yet have an instance of that response, you can use the `Cookie` facade to "queue" cookies for attachment to the response when it is sent. The `queue` method accepts the arguments needed to create a cookie instance. These cookies will be attached to the outgoing response before it is sent to the browser:

```
use Illuminate\Support\Facades\Cookie;

Cookie::queue('name', 'value', $minutes);
```

Generating Cookie Instances

If you would like to generate a `Symfony\Component\HttpFoundation\Cookie` instance that can be attached to a response instance at a later time, you may use the global `cookie` helper. This cookie will not be sent back to the client unless it is attached to a response instance:

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

Expiring Cookies Early

You may remove a cookie by expiring it via the `withoutCookie` method of an outgoing response:

```
return response('Hello World')->withoutCookie('name');
```

If you do not yet have an instance of the outgoing response, you may use the `Cookie` facade's `queue` method to expire a cookie:

```
Cookie::queue(Cookie::forget('name'));
```

Cookies & Encryption

By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client. If you would like to disable encryption for a subset of cookies generated by your application, you may use the `$except` property of the `App\Http\Middleware\EncryptCookies` middleware, which is located in the `app/Http/Middleware` directory:

```
/**
 * The names of the cookies that should not be encrypted.
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

Redirects

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

```
Route::get('/dashboard', function () {
    return redirect('home/dashboard');
});
```

Sometimes you may wish to redirect the user to their previous location, such as when a submitted form is invalid. You may do so by using the global `back` helper function. Since this feature utilizes the session, make sure the route calling the `back` function is using the `web` middleware group:

```
Route::post('/user/profile', function () {
    // Validate the request...

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

Redirecting To Named Routes

When you call the `redirect` helper with no parameters, an instance of `Illuminate\Routing\Redirector` is returned, allowing you to call any method on the `Redirector` instance. For example, to generate a `RedirectResponse` to a named route, you may use the `route` method:

```
return redirect()->route('login');
```

If your route has parameters, you may pass them as the second argument to the `route` method:

```
// For a route with the following URI: /profile/{id}
```

```
return redirect()->route('profile', ['id' => 1]);
```

Populating Parameters Via Eloquent Models

If you are redirecting to a route with an "ID" parameter that is being populated from an Eloquent model, you may pass the model itself. The ID will be extracted automatically:

```
// For a route with the following URI: /profile/{id}

return redirect()->route('profile', [$user]);
```

If you would like to customize the value that is placed in the route parameter, you can specify the column in the route parameter definition (`/profile/{id:slug}`) or you can override the `getRouteKey` method on your Eloquent model:

```
/**
 * Get the value of the model's route key.
 *
 * @return mixed
 */
public function getRouteKey()
{
    return $this->slug;
}
```

Redirecting To Controller Actions

You may also generate redirects to controller actions. To do so, pass the controller and action name to the `action` method:

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

return redirect()->action([UserController::class, 'index']);
```

If your controller route requires parameters, you may pass them as the second argument to the `action` method:

```
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

Redirecting To External Domains

Sometimes you may need to redirect to a domain outside of your application. You may do so by calling the `away` method, which creates a `RedirectResponse` without any additional URL encoding, validation, or verification:

```
return redirect()->away('https://www.google.com');
```

Redirecting With Flashed Session Data

Redirecting to a new URL and flashing data to the session are usually done at the same time. Typically, this is done after successfully performing an action when you flash a success message to the session. For convenience, you may create a `RedirectResponse` instance and flash data to the session in a single, fluent method chain:

```
Route::post('/user/profile', function () {
    // ...

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

After the user is redirected, you may display the flashed message from the session. For example, using Blade syntax:

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

Redirecting With Input

You may use the `withInput` method provided by the `RedirectResponse` instance to flash the current request's input data to the session before redirecting the user to a new location. This is typically done if the user has encountered a validation error. Once the input has been flashed to the session, you may easily retrieve it during the next request to repopulate the form:

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

Other Response Types

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

View Responses

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

```
return response()
            ->view('hello', $data, 200)
            ->header('Content-Type', $type);
```

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

JSON Responses

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

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

If you would like to create a JSONP response, you may use the `json` method in combination with the `withCallback` method:

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

File Downloads

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

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

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

> NOTE Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII filename.

Streamed Downloads

Sometimes you may wish to turn the string response of a given operation into a downloadable response without having to write the contents of the operation to disk. You may use the `streamDownload` method in this scenario. This method accepts a callback, filename, and an optional array of headers as its arguments:

```
use App\Services\GitHub;

return response()->streamDownload(function () {
    echo GitHub::api('repo')
                ->contents()
                ->readme('laravel', 'laravel')['contents'];
}, 'laravel-readme.md');
```

File Responses

The `file` method may be used to display a file, such as an image or PDF, directly in the user's browser instead of initiating a download. This method accepts the path to the file as its first argument and an array of headers as its second argument:

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

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

Response Macros

If you would like to define a custom response that you can re-use in a variety of your routes and controllers, you may use the `macro` method on the `Response` facade. Typically, you should call this method from the `boot` method of one of your application's service providers, such as the `App\Providers\AppServiceProvider` service provider:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

The `macro` function accepts a name as its first argument and a closure as its second argument. The macro's closure will be executed when calling the macro name from a `ResponseFactory` implementation or the `response` helper:

```
return response()->caps('foo');
```

The Basics

Views

- Introduction
- Creating & Rendering Views
  - Nested View Directories
  - Creating The First Available View
  - Determining If A View Exists
- Passing Data To Views
  - Sharing Data With All Views
- View Composers
  - View Creators
- Optimizing Views

Introduction

Of course, it's not practical to return entire HTML documents strings directly from your routes and controllers. Thankfully, views provide a convenient way to place all of our HTML in separate files. Views separate your controller / application logic from your presentation logic and are stored in the `resources/views` directory. A simple view might look something like this:

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

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Since this view is stored at `resources/views/greeting.blade.php`, we may return it using the global `view` helper like so:

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

> TIP Looking for more information on how to write Blade templates? Check out the full Blade documentation to get started.

Creating & Rendering Views

You may create a view by placing a file with the `.blade.php` extension in your application's `resources/views` directory. The `.blade.php` extension informs the framework that the file contains a Blade template. Blade templates contain HTML as well as Blade directives that allow you to easily echo values, create "if" statements, iterate over data, and more.

Once you have created a view, you may return it from one of your application's routes or controllers using the global `view` helper:

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

Views may also be returned using the `View` facade:

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

return View::make('greeting', ['name' => 'James']);
```

As you can see, the first argument passed to the `view` helper corresponds to the name of the view file in the `resources/views` directory. The second argument is an array of data that should be made available to the view. In this case, we are passing the `name` variable, which is displayed in the view using Blade syntax.

Nested View Directories

Views may also be nested within subdirectories of the `resources/views` directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at `resources/views/admin/profile.blade.php`, you may return it from one of your application's routes / controllers like so:

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

NOTE View directory names should not contain the `.` character.

Creating The First Available View

Using the `View` facade's `first` method, you may create the first view that exists in a given array of views. This may be useful if your application or package allows views to be customized or overwritten:

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

return View::first(['custom.admin', 'admin'], $data);
```

Determining If A View Exists

If you need to determine if a view exists, you may use the `View` facade. The `exists` method will return `true` if the view exists:

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

if (View::exists('emails.customer')) {
    //
}
```

Passing Data To Views

As you saw in the previous examples, you may pass an array of data to views to make that data available to the view:

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

When passing information in this manner, the data should be an array with key / value pairs. After providing data to a view, you can then access each value within your view using the data's keys, such as `<?php echo $name; ?>`.

As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view. The `with` method returns an instance of the view object so that you can continue chaining methods before returning the view:

```
return view('greeting')
            ->with('name', 'Victoria')
            ->with('occupation', 'Astronaut');
```

Sharing Data With All Views

Occasionally, you may need to share data with all views that are rendered by your application. You may do so using the `View` facade's `share` method. Typically, you should place calls to the `share` method within a service provider's `boot` method. You are free to add them to the `App\Providers\AppServiceProvider` class or generate a separate service provider to house them:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
```

```
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

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

View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location. View composers may prove particularly useful if the same view is returned by multiple routes or controllers within your application and always needs a particular piece of data.

Typically, view composers will be registered within one of your application's service providers. In this example, we'll assume that we have created a new `App\Providers\ViewServiceProvider` to house this logic.

We'll use the `View` facade's `composer` method to register the view composer. Laravel does not include a default directory for class based view composers, so you are free to organize them however you wish. For example, you could create an `app/Http/View/Composers` directory to house all of your application's view composers:

```
<?php

namespace App\Providers;

use App\Http\View\Composers\ProfileComposer;
use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

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

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

        // Using closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }
}
```

> NOTE Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the `providers` array in the `config/app.php` configuration file.

Now that we have registered the composer, the `compose` method of the `App\Http\View\Composers\ProfileComposer` class will be executed each time the `profile` view is being rendered. Let's take a look at an example of the composer class:

```
<?php

namespace App\Http\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * The user repository implementation.
     *
     * @var \App\Repositories\UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

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

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

Attaching A Composer To Multiple Views

You may attach a view composer to multiple views at once by passing an array of views as the first argument to the `composer` method:

```
use App\Http\Views\Composers\MultiComposer;

View::composer(
    ['profile', 'dashboard'],
    MultiComposer::class
);
```

The `composer` method also accepts the `*` character as a wildcard, allowing you to attach a composer to all views:

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

View Creators

View "creators" are very similar to view composers; however, they are executed immediately after the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the `creator` method:

```
use App\Http\View\Creators\ProfileCreator;
use Illuminate\Support\Facades\View;
```

```
View::creator('profile', ProfileCreator::class);
```

Optimizing Views

By default, Blade template views are compiled on demand. When a request is executed that renders a view, Laravel will determine if a compiled version of the view exists. If the file exists, Laravel will then determine if the uncompiled view has been modified more recently than the compiled view. If the compiled view either does not exist, or the uncompiled view has been modified, Laravel will recompile the view.

Compiling views during the request may have a small negative impact on performance, so Laravel provides the `view:cache` Artisan command to precompile all of the views utilized by your application. For increased performance, you may wish to run this command as part of your deployment process:

```
php artisan view:cache
```

You may use the `view:clear` command to clear the view cache:

```
php artisan view:clear
```

The Basics

Blade Templates

- Introduction
- Displaying Data
  - HTML Entity Encoding
  - Blade & JavaScript Frameworks
- Blade Directives
  - If Statements
  - Switch Statements
  - Loops
  - The Loop Variable
  - Comments
  - Including Subviews
  - The `@once` Directive
  - Raw PHP
- Components
  - Rendering Components
  - Passing Data To Components
  - Component Attributes
  - Slots
  - Inline Component Views
  - Anonymous Components
  - Dynamic Components
  - Manually Registering Components
- Building Layouts
  - Layouts Using Components
  - Layouts Using Template Inheritance
- Forms
  - CSRF Field
  - Method Field
  - Validation Errors
- Stacks
- Service Injection
- Extending Blade
  - Custom If Statements

Introduction

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Blade views may be returned from routes or controller using the global `view` helper. Of course, as mentioned in the documentation on views, data may be passed to the Blade view using the `view` helper's second argument:

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

TIP Before digging deeper into Blade, make sure to read the Laravel view documentation.

Displaying Data

You may display data that is passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

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

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

> TIP Blade's `{{ }}` echo statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

You are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

Rendering JSON

Sometimes you may pass an array to your view with the intention of rendering it as JSON in order to initialize a JavaScript variable. For example:

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

However, instead of manually calling `json_encode`, you may use the `@json` Blade directive. The `@json` directive accepts the same arguments as PHP's `json_encode` function. By default, the `@json` directive calls the `json_encode` function with the `JSON_HEX_TAG`, `JSON_HEX_APOS`, `JSON_HEX_AMP`, and `JSON_HEX_QUOT` flags:

```
<script>
    var app = @json($array);

    var app = @json($array, JSON_PRETTY_PRINT);
</script>
```

> NOTE You should only use the `@json` directive to render existing variables as JSON. The Blade templating is based on regular expressions and attempts to pass a complex expression to the directive may cause unexpected failures.

HTML Entity Encoding

By default, Blade (and the Laravel `e` helper) will double encode HTML entities. If you would like to disable double encoding, call the `Blade::withoutDoubleEncoding` method from the `boot` method of your `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
}
```

Displaying Unescaped Data

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

> NOTE Be very careful when echoing content that is supplied by users of your application. You should typically use the escaped, double curly brace syntax to prevent XSS attacks when displaying user supplied data.

Blade & JavaScript Frameworks

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

In this example, the `@` symbol will be removed by Blade; however, `{{ name }}` expression will remain untouched by the Blade engine, allowing it to be rendered by your JavaScript framework.

The `@` symbol may also be used to escape Blade directives:

```
{{-- Blade template --}}
@@json()

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

The `@verbatim` Directive

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the `@verbatim` directive so that you do not have to prefix each Blade echo statement with an `@` symbol:

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

Blade Directives

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

In addition to the conditional directives already discussed, the `@isset` and `@empty` directives may be used as convenient shortcuts for their respective PHP functions:

```
@isset($records)
```

```
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

Authentication Directives

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

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

If needed, you may specify the authentication guard that should be checked when using the `@auth` and `@guest` directives:

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

Environment Directives

You may check if the application is running in the production environment using the `@production` directive:

```
@production
    // Production specific content...
@endproduction
```

Or, you may determine if the application is running in a specific environment using the `@env` directive:

```
@env('staging')
    // The application is running in "staging"...
@endenv

@env(['staging', 'production'])
    // The application is running in "staging" or "production"...
@endenv
```

Section Directives

You may determine if a template inheritance section has content using the `@hasSection` directive:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

You may use the `sectionMissing` directive to determine if a section does not have content:

```
@sectionMissing('navigation')
    <div class="pull-right">
        @include('default-navigation')
    </div>
@endif
```

Switch Statements

Switch statements can be constructed using the `@switch`, `@case`, `@break`, `@default` and `@endswitch` directives:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

> TIP When looping, you may use the loop variable to gain valuable information about the loop, such as whether you are in the first or last iteration through the loop.

When using loops you may also end the loop or skip the current iteration using the `@continue` and `@break` directives:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

You may also include the continuation or break condition within the directive declaration:

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

The Loop Variable

When looping, a `$loop` variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

```
@foreach ($users as $user)
```

```
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

If you are in a nested loop, you may access the parent loop's `$loop` variable via the `parent` property:

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

The `$loop` variable also contains a variety of other useful properties:

Property	Description
`$loop->index`	The index of the current loop iteration (starts at 0).
`$loop->iteration`	The current loop iteration (starts at 1).
`$loop->remaining`	The iterations remaining in the loop.
`$loop->count`	The total number of items in the array being iterated.
`$loop->first`	Whether this is the first iteration through the loop.
`$loop->last`	Whether this is the last iteration through the loop.
`$loop->even`	Whether this is an even iteration through the loop.
`$loop->odd`	Whether this is an odd iteration through the loop.
`$loop->depth`	The nesting level of the current loop.
`$loop->parent`	When in a nested loop, the parent's loop variable.

Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Including Subviews

> TIP While you're free to use the `@include` directive, Blade components provide similar functionality and offer several benefits over the `@include` directive such as data and attribute binding.

Blade's `@include` directive allows you to include a Blade view from within another view. All variables that are available to the parent view will be made available to the included view:

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of additional data that should be made available to the included view:

```
@include('view.name', ['status' => 'complete'])
```

If you attempt to `@include` a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the `@includeIf` directive:

```
@includeIf('view.name', ['status' => 'complete'])
```

If you would like to `@include` a view if a given boolean expression evaluates to `true` or `false`, you may use the `@includeWhen` and `@includeUnless` directives:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])
```

```
@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

To include the first view that exists from a given array of views, you may use the `includeFirst` directive:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

> NOTE You should avoid using the `__DIR__` and `__FILE__` constants in your Blade views, since they will refer to the location of the cached, compiled view.

Rendering Views For Collections

You may combine loops and includes into one line with Blade's `@each` directive:

```
@each('view.name', $jobs, 'job')
```

The `@each` directive's first argument is the view to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of `jobs`, typically you will want to access each job as a `job` variable within the view. The array key for the current iteration will be available as the `key` variable within the view.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

> NOTE Views rendered via `@each` do not inherit the variables from the parent view. If the child view requires these variables, you should use the `@foreach` and `@include` directives instead.

The `@once` Directive

The `@once` directive allows you to define a portion of the template that will only be evaluated once per rendering cycle. This may be useful for pushing a given piece of JavaScript into the page's header using stacks. For example, if you are rendering a given component within a loop, you may wish to only push the JavaScript to the header the first time the component is rendered:

```
@once
    @push('scripts')
        <script>
            // Your custom JavaScript...
        </script>
    @endpush
@endonce
```

Building Layouts

Layouts Using Components

Most web applications maintain the same general layout across various pages. It would be incredibly cumbersome and hard to maintain our application if we had to repeat the entire layout HTML in every view we create. Thankfully, it's convenient to define this layout as a single Blade component and then use it throughout our application.

Defining The Layout Component

For example, imagine we are building a "todo" list application. We might define a `layout` component that looks like the following:

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

<html>
    <head>
        <title>{{ $title ?? 'Todo Manager' }}</title>
    </head>
    <body>
        <h1>Todos</h1>
        <hr/>
        {{ $slot }}
    </body>
</html>
```

Applying The Layout Component

Once the `layout` component has been defined, we may create a Blade view that utilizes the component. In this example, we will define a simple view that displays our task list:

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

<x-layout>
    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

Remember, content that is injected into a component will be supplied to the default `$slot` variable within our `layout` component. As you may have noticed, our `layout` also respects a `$title` slot if one is provided; otherwise, a default title is shown. We may inject a custom title from our task list view using the standard slot syntax discussed in the component documentation:

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

<x-layout>
    <x-slot name="title">
        Custom Title
    </x-slot>

    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

Now that we have defined our layout and task list views, we just need to return the `task` view from a route:

```
use App\Models\Task;

Route::get('/tasks', function () {
    return view('tasks', ['tasks' => Task::all()]);
});
```

Layouts Using Template Inheritance

Defining A Layout

Layouts may also be created via "template inheritance". This was the primary way of building applications prior to the introduction of components.

To get started, let's take a look at a simple example. First, we will examine a page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show
```

```
        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

Extending A Layout

When defining a child view, use the `@extends` Blade directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

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

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

In this example, the `sidebar` section is utilizing the `@@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@@parent` directive will be replaced by the content of the layout when the view is rendered.

> TIP Contrary to the previous example, this `sidebar` section ends with `@endsection` instead of `@show`. The `@endsection` directive will only define a section while `@show` will define and **immediately yield** the section.

The `@yield` directive also accepts a default value as its second parameter. This value will be rendered if the section being yielded is undefined:

```
@yield('content', 'Default content')
```

Forms

CSRF Field

Anytime you define an HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the `@csrf` Blade directive to generate the token field:

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

    ...
</form>
```

Method Field

Since HTML forms can't make `PUT`, `PATCH`, or `DELETE` requests, you will need to add a hidden `_method` field to spoof these HTTP verbs. The `@method` Blade directive can create this field for you:

```
<form action="/foo/bar" method="POST">
    @method('PUT')

    ...
</form>
```

Validation Errors

The `@error` directive may be used to quickly check if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

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

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

You may pass the name of a specific error bag as the second parameter to the `@error` directive to retrieve validation error messages on pages containing multiple forms:

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

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Raw PHP

In some situations, it's useful to embed PHP code into your views. You can use the Blade `@php` directive to execute a block of plain PHP within your template:

```
@php
    $counter = 1;
@endphp
```

Components

Components and slots provide similar benefits to sections, layouts, and includes; however, some may find the mental model of components and slots easier to understand. There are two approaches to writing components: class based components and anonymous components.

To create a class based component, you may use the `make:component` Artisan command. To illustrate how to use components, we will create a simple `Alert` component. The `make:component` command will place the component in the `App\View\Components` directory:

```
php artisan make:component Alert
```

The `make:component` command will also create a view template for the component. The view will be placed in the `resources/views/components` directory. When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory, so no further component registration is typically required.

You may also create components within subdirectories:

```
php artisan make:component Forms/Input
```

The command above will create an `Input` component in the `App\View\Components\Forms` directory and the view will be placed in the `resources/views/components/forms` directory.

Manually Registering Package Components

When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory.

However, if you are building a package that utilizes Blade components, you will need to manually register your component class and its HTML tag alias. You should typically register your components in the `boot` method of your package's service provider:

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

/**
 * Bootstrap your package's services.
 */
public function boot()
{
    Blade::component('package-alert', AlertComponent::class);
}
```

Once your component has been registered, it may be rendered using its tag alias:

```
<x-package-alert/>
```

Alternatively, you may use the `componentNamespace` method to autoload component classes by convention. For example, a `Nightshade` package might have `Calendar` and `ColorPicker` components that reside within the `Package\Views\Components` namespace:

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

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

This will allow the usage of package components by their vendor namespace using the `package-name::` syntax:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

Rendering Components

To display a component, you may use a Blade component tag within one of your Blade templates. Blade component tags start with the string `x-` followed by the kebab case name of the component class:

```
<x-alert/>

<x-user-profile/>
```

If the component class is nested deeper within the `App\View\Components` directory, you may use the `.` character to indicate directory nesting. For example, if we assume a component is located at `App\View\Components\Inputs\Button.php`, we may render it like so:

```
<x-inputs.button/>
```

Passing Data To Components

You may pass data to Blade components using HTML attributes. Hard-coded, primitive values may be passed to the component using simple HTML attribute strings. PHP expressions and variables should be passed to the component via attributes that use the `:` character as a prefix:

```
<x-alert type="error" :message="$message"/>
```

You should define the component's required data in its class constructor. All public properties on a component will automatically be made available to the component's view. It is not necessary to pass the data to the view from the component's `render` method:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
    /**
     * The alert type.
     *
     * @var string
     */
    public $type;

    /**
     * The alert message.
     *
     * @var string
     */
    public $message;

    /**
     * Create the component instance.
     *
     * @param  string  $type
     * @param  string  $message
     * @return void
     */
    public function __construct($type, $message)
    {
        $this->type = $type;
        $this->message = $message;
    }

    /**
     * Get the view / contents that represent the component.
     *
     * @return \Illuminate\View\View|\Closure|string
     */
    public function render()
    {
        return view('components.alert');
    }
}
```

When your component is rendered, you may display the contents of your component's public variables by echoing the variables by name:

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

Casing

Component constructor arguments should be specified using `camelCase`, while `kebab-case` should be used when referencing the argument names in your HTML attributes. For example, given the following component constructor:

```
/**
 * Create the component instance.
 *
 * @param  string  $alertType
 * @return void
 */
public function __construct($alertType)
{
    $this->alertType = $alertType;
}
```

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

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

Component Methods

In addition to public variables being available to your component template, any public methods on the component may be invoked. For example, imagine a component that has a `isSelected` method:

```
/**
 * Determine if the given option is the current selected option.
 *
 * @param  string  $option
 * @return bool
 */
public function isSelected($option)
{
    return $option === $this->selected;
}
```

You may execute this method from your component template by invoking the variable matching the name of the method:

```
<option {{ $isSelected($value) ? 'selected="selected"' : '' }} value="{{ $value }}">
    {{ $label }}
</option>
```

Accessing Attributes & Slots Within Component Classes

Blade components also allow you to access the component name, attributes, and slot inside the class's render method. However, in order to access this data, you should return a closure from your component's `render` method. The closure will receive a `$data` array as its only argument. This array will contain several elements that provide information about the component:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
    return function (array $data) {
        // $data['componentName'];
        // $data['attributes'];
        // $data['slot'];

        return '<div>Components content</div>';
    };
}
```

The `componentName` is equal to the name used in the HTML tag after the `x-` prefix. So `<x-alert />`'s `componentName` will be `alert`. The `attributes` element will contain all of the attributes that were present on the HTML tag. The `slot` element is an `Illuminate\Support\HtmlString` instance with the contents of the component's slot.

The closure should return a string. If the returned string corresponds to an existing view, that view will be rendered; otherwise, the returned string will be evaluated as an inline Blade view.

Additional Dependencies

If your component requires dependencies from Laravel's service container, you may list them before any of the component's data attributes and they will automatically be injected by the container:

```
use App\Services\AlertCreator

/**
 * Create the component instance.
 *
 * @param  \App\Services\AlertCreator  $creator
 * @param  string  $type
```

```
 * @param  string  $message
 * @return void
 */
public function __construct(AlertCreator $creator, $type, $message)
{
    $this->creator = $creator;
    $this->type = $type;
    $this->message = $message;
}
```

Component Attributes

We've already examined how to pass data attributes to a component; however, sometimes you may need to specify additional HTML attributes, such as `class`, that are not part of the data required for a component to function. Typically, you want to pass these additional attributes down to the root element of the component template. For example, imagine we want to render an `alert` component like so:

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

All of the attributes that are not part of the component's constructor will automatically be added to the component's "attribute bag". This attribute bag is automatically made available to the component via the `$attributes` variable. All of the attributes may be rendered within the component by echoing this variable:

```
<div {{ $attributes }}>
    <!-- Component content -->
</div>
```

> NOTE Using directives such as `@env` within component tags is not supported at this time. For example, `<x-alert :live="@env('production')"/>` will not be compiled.

Default / Merged Attributes

Sometimes you may need to specify default values for attributes or merge additional values into some of the component's attributes. To accomplish this, you may use the attribute bag's `merge` method. This method is particularly useful for defining a set a default CSS classes that should always be applied to a component:

```
<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    {{ $message }}
</div>
```

If we assume this component is utilized like so:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

The final, rendered HTML of the component will appear like the following:

```
<div class="alert alert-error mb-4">
    <!-- Contents of the $message variable -->
</div>
```

Non-Class Attribute Merging

When merging attributes that are not `class` attributes, the values provided to the `merge` method will be considered the "default" values of attribute. However, unlike the `class` attribute, these attributes will not be merged with injected attribute values. Instead, they will be overwritten. For example, a `button` component's implementation may look like the following:

```
<button {{ $attributes->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

To render the button component with a custom `type`, it may be specified when consuming the component. If no type is specified, the `button` type will be used:

```
<x-button type="submit">
    Submit
</x-button>
```

The rendered HTML of the `button` component in this example would be:

```
<button type="submit">
    Submit
</button>
```

If you would like an attribute other than `class` to have its default value and injected values joined together, you may use the `prepends` method. In this example, the `data-controller` attribute will always begin with `profile-controller` and any additional injected `data-controller` values will be placed after this default value:

```
<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-controller')]) }}>
    {{ $slot }}
</div>
```

Retrieving & Filtering Attributes

You may filter attributes using the `filter` method. This method accepts a closure which should return `true` if you wish to retain the attribute in the attribute bag:

```
{{ $attributes->filter(fn ($value, $key) => $key == 'foo') }}
```

For convenience, you may use the `whereStartsWith` method to retrieve all attributes whose keys begin with a given string:

```
{{ $attributes->whereStartsWith('wire:model') }}
```

Using the `first` method, you may render the first attribute in a given attribute bag:

```
{{ $attributes->whereStartsWith('wire:model')->first() }}
```

If you would like to check if an attribute is present on the component, you may use the `has` method. This method accepts the attribute name as its only argument and returns a boolean indicating whether or not the attribute is present:

```
@if ($attributes->has('class'))
    <div>Class attribute is present</div>
@endif
```

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

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

Slots

You will often need to pass additional content to your component via "slots". Component slots are rendered by echoing the `$slot` variable. To explore this concept, let's imagine that an `alert` component has the following markup:

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

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

We may pass content to the `slot` by injecting content into the component:

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

Sometimes a component may need to render multiple different slots in different locations within the component. Let's modify our alert component to allow for the injection of a "title" slot:

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

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

<div class="alert alert-danger">
    {{ $slot }}
```

```
</div>
```

You may define the content of the named slot using the `x-slot` tag. Any content not within an explicit `x-slot` tag will be passed to the component in the `$slot` variable:

```
<x-alert>
    <x-slot name="title">
        Server Error
    </x-slot>

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

Scoped Slots

If you have used a JavaScript framework such as Vue, you may be familiar with "scoped slots", which allow you to access data or methods from the component within your slot. You may achieve similar behavior in Laravel by defining public methods or properties on your component and accessing the component within your slot via the `$component` variable. In this example, we will assume that the `x-alert` component has a public `formatAlert` method defined on its component class:

```
<x-alert>
    <x-slot name="title">
        {{ $component->formatAlert('Server Error') }}
    </x-slot>

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

Inline Component Views

For very small components, it may feel cumbersome to manage both the component class and the component's view template. For this reason, you may return the component's markup directly from the `render` method:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    blade;
}
```

Generating Inline View Components

To create a component that renders an inline view, you may use the `inline` option when executing the `make:component` command:

```
php artisan make:component Alert --inline
```

Anonymous Components

Similar to inline components, anonymous components provide a mechanism for managing a component via a single file. However, anonymous components utilize a single view file and have no associated class. To define an anonymous component, you only need to place a Blade template within your `resources/views/components` directory. For example, assuming you have defined a component at `resources/views/components/alert.blade.php`, you may simply render it like so:

```
<x-alert/>
```

You may use the `.` character to indicate if a component is nested deeper inside the `components` directory. For example, assuming the component is defined at `resources/views/components/inputs/button.blade.php`, you may render it like so:

```
<x-inputs.button/>
```

Data Properties / Attributes

Since anonymous components do not have any associated class, you may wonder how you may differentiate which data should be passed to the component as variables and which attributes should be placed in the component's attribute bag.

You may specify which attributes should be considered data variables using the `@props` directive at the top of your component's Blade template. All other attributes on the component will be available via the component's attribute bag. If you wish to give a data variable a default value, you may specify the variable's name as the array key and the default value as the array value:

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

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    {{ $message }}
</div>
```

Given the component definition above, we may render the component like so:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Dynamic Components

Sometimes you may need to render a component but not know which component should be rendered until runtime. In this situation, you may use Laravel's built-in `dynamic-component` component to render the component based on a runtime value or variable:

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

Manually Registering Components

> NOTE The following documentation on manually registering components is primarily applicable to those who are writing Laravel packages that include view components. If you are not writing a package, this portion of the component documentation may not be relevant to you.

When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory.

However, if you are building a package that utilizes Blade components or placing components in non-conventional directories, you will need to manually register your component class and its HTML tag alias so that Laravel knows where to find the component. You should typically register your components in the `boot` method of your package's service provider:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::component('package-alert', AlertComponent::class);
}
```

Once your component has been registered, it may be rendered using its tag alias:

```
<x-package-alert/>
```

Autoloading Package Components

Alternatively, you may use the `componentNamespace` method to autoload component classes by convention. For example, a `Nightshade` package might have `Calendar` and `ColorPicker` components that reside within the `Package\Views\Components` namespace:

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

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

This will allow the usage of package components by their vendor namespace using the `package-name::` syntax:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

Stacks

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the `@stack` directive:

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

If you would like to prepend content onto the beginning of a stack, you should use the `@prepend` directive:

```
@push('scripts')
    This will be second...
@endpush

// Later...

@prepend('scripts')
    This will be first...
@endprepend
```

Service Injection

The `@inject` directive may be used to retrieve a service from the Laravel service container. The first argument passed to `@inject` is the name of the variable the service will be placed into, while the second argument is the class or interface name of the service you wish to resolve:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

Extending Blade

Blade allows you to define your own custom directives using the `directive` method. When the Blade compiler

encounters the custom directive, it will call the provided callback with the expression that the directive contains.

The following example creates a `@datetime($var)` directive which formats a given `$var`, which should be an instance of `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

As you can see, we will chain the `format` method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

> NOTE After updating the logic of a Blade directive, you will need to delete all of the cached Blade views. The cached Blade views may be removed using the `view:clear` Artisan command.

Custom If Statements

Programming a custom directive is sometimes more complex than necessary when defining simple, custom conditional statements. For that reason, Blade provides a `Blade::if` method which allows you to quickly define custom conditional directives using closures. For example, let's define a custom conditional that checks the configured default "disk" for the application. We may do this in the `boot` method of our `AppServiceProvider`:

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Blade::if('disk', function ($value) {
        return config('filesystems.default') === $value;
    });
}
```

Once the custom conditional has been defined, you can use it within your templates:

```
@disk('local')
    <!-- The application is using the local disk... -->
@elsedisk('s3')
    <!-- The application is using the s3 disk... -->
@else
```

```
    <!-- The application is using some other disk... -->
@enddisk

@unlessdisk('local')
    <!-- The application is not using the local disk... -->
@enddisk
```

The Basics

URL Generation

- Introduction
- The Basics
  - Generating URLs
  - Accessing The Current URL
- URLs For Named Routes
  - Signed URLs
- URLs For Controller Actions
- Default Values

Introduction

Laravel provides several helpers to assist you in generating URLs for your application. These helpers are primarily helpful when building links in your templates and API responses, or when generating redirect responses to another part of your application.

The Basics

Generating URLs

The `url` helper may be used to generate arbitrary URLs for your application. The generated URL will automatically use the scheme (HTTP or HTTPS) and host from the current request being handled by the application:

```
$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1
```

Accessing The Current URL

If no path is provided to the `url` helper, an `Illuminate\Routing\UrlGenerator` instance is returned, allowing you to access information about the current URL:

```
// Get the current URL without the query string...
echo url()->current();

// Get the current URL including the query string...
echo url()->full();

// Get the full URL for the previous request...
echo url()->previous();
```

Each of these methods may also be accessed via the `URL` facade:

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

echo URL::current();
```

URLs For Named Routes

The `route` helper may be used to generate URLs to named routes. Named routes allow you to generate URLs without being coupled to the actual URL defined on the route. Therefore, if the route's URL changes, no changes need to be made to your calls to the `route` function. For example, imagine your application contains a route defined like the following:

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

To generate a URL to this route, you may use the `route` helper like so:

```
echo route('post.show', ['post' => 1]);

// http://example.com/post/1
```

Of course, the `route` helper may also be used to generate URLs for routes with multiple parameters:

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

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3
```

Any additional array elements that do not correspond to the route's definition parameters will be added to the URL's query string:

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);

// http://example.com/post/1?search=rocket
```

Eloquent Models

You will often be generating URLs using the primary key of Eloquent models. For this reason, you may pass Eloquent models as parameter values. The `route` helper will automatically extract the model's primary key:

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

Signed URLs

Laravel allows you to easily create "signed" URLs to named routes. These URLs have a "signature" hash appended to the query string which allows Laravel to verify that the URL has not been modified since it was created. Signed URLs are especially useful for routes that are publicly accessible yet need a layer of protection against URL manipulation.

For example, you might use signed URLs to implement a public "unsubscribe" link that is emailed to your customers. To create a signed URL to a named route, use the `signedRoute` method of the `URL` facade:

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

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

If you would like to generate a temporary signed route URL that expires after a specified amount of time, you may use the `temporarySignedRoute` method. When Laravel validates a temporary signed route URL, it will ensure that the expiration timestamp that is encoded into the signed URL has not elapsed:

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

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

Validating Signed Route Requests

To verify that an incoming request has a valid signature, you should call the `hasValidSignature` method on the incoming `Request`:

```
use Illuminate\Http\Request;

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

    // ...
})->name('unsubscribe');
```

Alternatively, you may assign the `Illuminate\Routing\Middleware\ValidateSignature` middleware to the route. If it is not already present, you should assign this middleware a key in your HTTP kernel's `routeMiddleware` array:

```
/**
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
];
```

Once you have registered the middleware in your kernel, you may attach it to a route. If the incoming request does not have a valid signature, the middleware will automatically return a `403` HTTP response:

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

URLs For Controller Actions

The `action` function generates a URL for the given controller action:

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

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

If the controller method accepts route parameters, you may pass an associative array of route parameters as the second argument to the function:

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

Default Values

For some applications, you may wish to specify request-wide default values for certain URL parameters. For example, imagine many of your routes define a `{locale}` parameter:

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

It is cumbersome to always pass the `locale` every time you call the `route` helper. So, you may use the `URL::defaults` method to define a default value for this parameter that will always be applied during the current request. You may wish to call this method from a route middleware so that you have access to the current request:

```
<?php

namespace App\Http\Middleware;

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

class SetDefaultLocaleForUrls
{
    /**
     * Handle the incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return \Illuminate\Http\Response
     */
    public function handle($request, Closure $next)
    {
        URL::defaults(['locale' => $request->user()->locale]);

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

Once the default value for the `locale` parameter has been set, you are no longer required to pass its value when generating URLs via the `route` helper.

URL Defaults & Middleware Priority

Setting URL default values can interfere with Laravel's handling of implicit model bindings. Therefore, you should prioritize your middleware that set URL defaults to be executed before Laravel's own `SubstituteBindings` middleware. You can accomplish this by making sure your middleware occurs before the `SubstituteBindings` middleware within the `$middlewarePriority` property of your application's HTTP kernel.

The `$middlewarePriority` property is defined in the base `Illuminate\Foundation\Http\Kernel` class. You may copy its definition from that class and overwrite it in your application's HTTP kernel in order to modify it:

```
/**
 * The priority-sorted list of middleware.
 *
 * This forces non-global middleware to always be in the given order.
 *
 * @var array
 */
protected $middlewarePriority = [
    // ...
     \App\Http\Middleware\SetDefaultLocaleForUrls::class,
     \Illuminate\Routing\Middleware\SubstituteBindings::class,
     // ...
];
```

The Basics

HTTP Session

- Introduction
  - Configuration
  - Driver Prerequisites
- Interacting With The Session
  - Retrieving Data
  - Storing Data
  - Flash Data
  - Deleting Data
  - Regenerating The Session ID
- Session Blocking
- Adding Custom Session Drivers
  - Implementing The Driver
  - Registering The Driver

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. That user information is typically placed in a persistent store / backend that can be accessed from subsequent requests.

Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as Memcached, Redis, and databases is included.

Configuration

Your application's session configuration file is stored at `config/session.php`. Be sure to review the options available to you in this file. By default, Laravel is configured to use the `file` session driver, which will work well for many applications. If your application will be load balanced across multiple web servers, you should choose a centralized store that all servers can access, such as Redis or a database.

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

- `file` - sessions are stored in `storage/framework/sessions`.
- `cookie` - sessions are stored in secure, encrypted cookies.
- `database` - sessions are stored in a relational database.
- `memcached` / `redis` - sessions are stored in one of these fast, cache based stores.
- `dynamodb` - sessions are stored in AWS DynamoDB.
- `array` - sessions are stored in a PHP array and will not be persisted.

> TIP The array driver is primarily used during testing and prevents the data stored in the session from being persisted.

Driver Prerequisites

Database

When using the `database` session driver, you will need to create a table to contain the session records. An example `Schema` declaration for the table may be found below:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->primary();
    $table->foreignId('user_id')->nullable()->index();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
```

```
    $table->integer('last_activity')->index();
});
```

You may use the `session:table` Artisan command to generate this migration. To learn more about database migrations, you may consult the complete migration documentation:

```
php artisan session:table

php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. For more information on configuring Redis, consult Laravel's Redis documentation.

> TIP In the `session` configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

Interacting With The Session

Retrieving Data

There are two primary ways of working with session data in Laravel: the global `session` helper and via a `Request` instance. First, let's look at accessing the session via a `Request` instance, which can be type-hinted on a route closure or controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

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

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

        //
    }
}
```

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

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

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

The Global Session Helper

You may also use the global `session` PHP function to retrieve and store data in the session. When the `session` helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

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

    // Specifying a default value...
    $value = session('key', 'default');

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

TIP There is little practical difference between using the session via an HTTP request instance versus using the global `session` helper. Both methods are testable via the `assertSessionHas` method which is available in all of your test cases.

Retrieving All Session Data

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

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

Determining If An Item Exists In The Session

To determine if an item is present in the session, you may use the `has` method. The `has` method returns `true` if the item is present and is not `null`:

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

To determine if an item is present in the session, even if its value is `null`, you may use the `exists` method:

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

Storing Data

To store data in the session, you will typically use the request instance's `put` method or the `session` helper:

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global "session" helper...
session(['key' => 'value']);
```

Pushing To Array Session Values

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

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

Retrieving & Deleting An Item

The `pull` method will retrieve and delete an item from the session in a single statement:

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

Flash Data

Sometimes you may wish to store items in the session for the next request. You may do so using the `flash` method. Data stored in the session using this method will be available immediately and during the subsequent HTTP request. After the subsequent HTTP request, the flashed data will be deleted. Flash data is primarily useful for short-lived status messages:

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

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

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

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

Deleting Data

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

```
// Forget a single key...
$request->session()->forget('name');

// Forget multiple keys...
$request->session()->forget(['name', 'status']);

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

Regenerating The Session ID

Regenerating the session ID is often done in order to prevent malicious users from exploiting a session fixation attack on your application.

Laravel automatically regenerates the session ID during authentication if you are using one of the Laravel application starter kits or Laravel Fortify; however, if you need to manually regenerate the session ID, you may use the `regenerate` method.

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

Session Blocking

> NOTE To utilize session blocking, your application must be using a cache driver that supports atomic locks. Currently, those cache drivers include the `memcached`, `dynamodb`, `redis`, and `database` drivers. In addition, you may not use the `cookie` session driver.

By default, Laravel allows requests using the same session to execute concurrently. So, for example, if you use a JavaScript HTTP library to make two HTTP requests to your application, they will both execute at the same time. For many applications, this is not a problem; however, session data loss can occur in a small subset of applications that make concurrent requests to two different application endpoints which both write data to the session.

To mitigate this, Laravel provides functionality that allows you to limit concurrent requests for a given session. To get started, you may simply chain the `block` method onto your route definition. In this example, an incoming request to the `/profile` endpoint would acquire a session lock. While this lock is being held, any incoming requests to the `/profile` or `/order` endpoints which share the same session ID will wait for the first request to finish executing before continuing their execution:

```
Route::post('/profile', function () {
    //
})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {
    //
})->block($lockSeconds = 10, $waitSeconds = 10)
```

The `block` method accepts two optional arguments. The first argument accepted by the `block` method is the maximum number of seconds the session lock should be held for before it is released. Of course, if the request finishes executing before this time the lock will be released earlier.

The second argument accepted by the `block` method is the number of seconds a request should wait while

attempting to obtain a session lock. An `Illuminate\Contracts\Cache\LockTimeoutException` will be thrown if the request is unable to obtain a session lock within the given number of seconds.

If neither of these arguments are passed, the lock will be obtained for a maximum of 10 seconds and requests will wait a maximum of 10 seconds while attempting to obtain a lock:

```
Route::post('/profile', function () {
    //
})->block()
```

Adding Custom Session Drivers

Implementing The Driver

If none of the existing session drivers fit your application's needs, Laravel makes it possible to write your own session handler. Your custom session driver should implement PHP's built-in `SessionHandlerInterface`. This interface contains just a few simple methods. A stubbed MongoDB implementation looks like the following:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

> TIP Laravel does not ship with a directory to contain your extensions. You are free to place them anywhere you like. In this example, we have created an `Extensions` directory to house the `MongoSessionHandler`.

Since the purpose of these methods is not readily understandable, let's quickly cover what each of the methods do:

- The `open` method would typically be used in file based session store systems. Since Laravel ships with a `file` session driver, you will almost never need to put anything in this method. You can simply leave this method empty.
- The `close` method, like the `open` method, can also usually be disregarded. For most drivers, it is not needed.
- The `read` method should return the string version of the session data associated with the given `$sessionId`. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
- The `write` method should write the given `$data` string associated with the `$sessionId` to some persistent storage system, such as MongoDB or another storage system of your choice. Again, you should not perform any serialization - Laravel will have already handled that for you.
- The `destroy` method should remove the data associated with the `$sessionId` from persistent storage.
- The `gc` method should destroy all session data that is older than the given `$lifetime`, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

Registering The Driver

Once your driver has been implemented, you are ready to register it with Laravel. To add additional drivers to Laravel's session backend, you may use the `extend` method provided by the `Session` facade. You should call the `extend` method from the `boot` method of a service provider. You may do this from the existing `App\Providers\AppServiceProvider` or create an entirely new provider:

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Support\Facades\Session;
```

```
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return an implementation of SessionHandlerInterface...
            return new MongoSessionHandler;
        });
    }
}
```

Once the session driver has been registered, you may use the `mongo` driver in your `config/session.php` configuration file.

The Basics

Validation

- Introduction
- Validation Quickstart
  - Defining The Routes
  - Creating The Controller
  - Writing The Validation Logic
  - Displaying The Validation Errors
  - Repopulating Forms
  - A Note On Optional Fields
- Form Request Validation
  - Creating Form Requests
  - Authorizing Form Requests
  - Customizing The Error Messages
  - Preparing Input For Validation
- Manually Creating Validators
  - Automatic Redirection
  - Named Error Bags
  - Customizing The Error Messages
  - After Validation Hook
- Working With Error Messages
  - Specifying Custom Messages In Language Files
  - Specifying Attributes In Language Files
  - Specifying Values In Language Files
- Available Validation Rules
- Conditionally Adding Rules
- Validating Arrays
- Custom Validation Rules
  - Using Rule Objects
  - Using Closures
  - Implicit Rules

Introduction

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the `validate` method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user. By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:

Defining The Routes

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

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

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

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

Creating The Controller

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

```
<?php

namespace App\Http\Controllers;

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

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

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

Writing The Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. To do this, we will use the `validate` method provided by the `Illuminate\Http\Request` object. If the validation rules pass, your code will keep executing normally; however, if validation fails, an exception will be thrown and the proper error response will automatically be sent back to the user.

If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

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

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

    // The blog post is valid...
}
```

As you can see, the validation rules are passed into the `validate` method. Don't worry - all available validation rules are documented. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Alternatively, validation rules may be specified as arrays of rules instead of a single `|` delimited string:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

In addition, you may use the `validateWithBag` method to validate a request and store any error messages within a named error bag:

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Stopping On First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

In this example, if the `unique` rule on the `title` attribute fails, the `max` rule will not be checked. Rules will be validated in the order they are assigned.

A Note On Nested Attributes

If the incoming HTTP request contains "nested" field data, you may specify these fields in your validation rules using "dot" syntax:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

On the other hand, if your field name contains a literal period, you can explicitly prevent this from being interpreted as "dot" syntax by escaping the period with a backslash:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'v1\.0' => 'required',
]);
```

Displaying The Validation Errors

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

An `$errors` variable is shared with all of your application's views by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. When this middleware is applied an `$errors` variable will always be available in your views, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

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

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

<h1>Create Post</h1>

@if ($errors->any())
```

```
[Accepted](#rule-accepted)
[Active URL](#rule-active-url)
[After (Date)](#rule-after)
[After Or Equal (Date)](#rule-after-or-equal)
[Alpha](#rule-alpha)
[Alpha Dash](#rule-alpha-dash)
[Alpha Numeric](#rule-alpha-num)
[Array](#rule-array)
[Bail](#rule-bail)
[Before (Date)](#rule-before)
[Before Or Equal (Date)](#rule-before-or-equal)
[Between](#rule-between)
[Boolean](#rule-boolean)
[Confirmed](#rule-confirmed)
[Date](#rule-date)
[Date Equals](#rule-date-equals)
[Date Format](#rule-date-format)
[Different](#rule-different)
[Digits](#rule-digits)
[Digits Between](#rule-digits-between)
[Dimensions (Image Files)](#rule-dimensions)
[Distinct](#rule-distinct)
[Email](#rule-email)
[Ends With](#rule-ends-with)
[Exclude If](#rule-exclude-if)
[Exclude Unless](#rule-exclude-unless)
[Exists (Database)](#rule-exists)
[File](#rule-file)
[Filled](#rule-filled)
[Greater Than](#rule-gt)
[Greater Than Or Equal](#rule-gte)
[Image (File)](#rule-image)
[In](#rule-in)
[In Array](#rule-in-array)
[Integer](#rule-integer)
[IP Address](#rule-ip)
[JSON](#rule-json)
[Less Than](#rule-lt)
[Less Than Or Equal](#rule-lte)
[Max](#rule-max)
[MIME Types](#rule-mimetypes)
[MIME Type By File Extension](#rule-mimes)
[Min](#rule-min)
[Multiple Of](#multiple-of)
[Not In](#rule-not-in)
[Not Regex](#rule-not-regex)
[Nullable](#rule-nullable)
[Numeric](#rule-numeric)
[Password](#rule-password)
[Present](#rule-present)
[Regular Expression](#rule-regex)
[Required](#rule-required)
[Required If](#rule-required-if)
[Required Unless](#rule-required-unless)
[Required With](#rule-required-with)
[Required With All](#rule-required-with-all)
[Required Without](#rule-required-without)
[Required Without All](#rule-required-without-all)
[Same](#rule-same)
[Size](#rule-size)
[Sometimes](#conditionally-adding-rules)
[Starts With](#rule-starts-with)
[String](#rule-string)
[Timezone](#rule-timezone)
[Unique (Database)](#rule-unique)
[URL](#rule-url)
[UUID](#rule-uuid)


<a name="rule-accepted"></a>
#### accepted

The field under validation must be `"yes"`, `"on"`, `1`, or `true`. This is useful for validating "Terms
of Service" acceptance or similar fields.

<a name="rule-active-url"></a>
#### active_url
```

```
The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP
function. The hostname of the provided URL is extracted using the `parse_url` PHP function before being
passed to `dns_get_record`.

<a name="rule-after"></a>
#### after:_date_

The field under validation must be a value after a given date. The dates will be passed into the
`strtotime` PHP function in order to be converted to a valid `DateTime` instance:

    'start_date' => 'required|date|after:tomorrow'

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to
compare against the date:

    'finish_date' => 'required|date|after:start_date'

<a name="rule-after-or-equal"></a>
#### after\_or\_equal:_date_

The field under validation must be a value after or equal to the given date. For more information, see
the [after](#rule-after) rule.

<a name="rule-alpha"></a>
#### alpha

The field under validation must be entirely alphabetic characters.

<a name="rule-alpha-dash"></a>
#### alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

<a name="rule-alpha-num"></a>
#### alpha_num

The field under validation must be entirely alpha-numeric characters.

<a name="rule-array"></a>
#### array

The field under validation must be a PHP `array`.

<a name="rule-bail"></a>
#### bail

Stop running validation rules after the first validation failure.

<a name="rule-before"></a>
#### before:_date_

The field under validation must be a value preceding the given date. The dates will be passed into the
PHP `strtotime` function in order to be converted into a valid `DateTime` instance. In addition, like
the [`after`](#rule-after) rule, the name of another field under validation may be supplied as the value
of `date`.

<a name="rule-before-or-equal"></a>
#### before\_or\_equal:_date_

The field under validation must be a value preceding or equal to the given date. The dates will be
passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance. In
addition, like the [`after`](#rule-after) rule, the name of another field under validation may be
supplied as the value of `date`.

<a name="rule-between"></a>
#### between:_min_,_max_

The field under validation must have a size between the given _min_ and _max_. Strings, numerics,
arrays, and files are evaluated in the same fashion as the [`size`](#rule-size) rule.

<a name="rule-boolean"></a>
#### boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`,
`1`, `0`, `"1"`, and `"0"`.

<a name="rule-confirmed"></a>
#### confirmed
```

The field under validation must have a matching field of `{field}_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

<a name="rule-date"></a>
date

The field under validation must be a valid, non-relative date according to the `strtotime` PHP function.

<a name="rule-date-equals"></a>
date_equals:_date_

The field under validation must be equal to the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance.

<a name="rule-date-format"></a>
date_format:_format_

The field under validation must match the given _format_. You should use **either** `date` or `date_format` when validating a field, not both. This validation rule supports all formats supported by PHP's [DateTime](https://www.php.net/manual/en/class.datetime.php) class.

<a name="rule-different"></a>
different:_field_

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

<a name="rule-digits"></a>
digits:_value_

The field under validation must be _numeric_ and must have an exact length of _value_.

<a name="rule-digits-between"></a>
digits_between:_min_,_max_

The field under validation must be _numeric_ and must have a length between the given _min_ and _max_.

<a name="rule-dimensions"></a>
dimensions

The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters:

```
    'avatar' => 'dimensions:min_width=100,min_height=200'
```

Available constraints are: _min_width_, _max_width_, _min_height_, _max_height_, _width_, _height_, _ratio_.

A _ratio_ constraint should be represented as width divided by height. This can be specified either by a fraction like `3/2` or a float like `1.5`:

```
    'avatar' => 'dimensions:ratio=3/2'
```

Since this rule requires several arguments, you may use the `Rule::dimensions` method to fluently construct the rule:

```
    use Illuminate\Support\Facades\Validator;
    use Illuminate\Validation\Rule;

    Validator::make($data, [
        'avatar' => [
            'required',
            Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
        ],
    ]);
```

<a name="rule-distinct"></a>
distinct

When validating arrays, the field under validation must not have any duplicate values:

```
    'foo.*.id' => 'distinct'
```

You may add `ignore_case` to the validation rule's arguments to make the rule ignore capitalization differences:

```
    'foo.*.id' => 'distinct:ignore_case'
```

<a name="rule-email"></a>
email

The field under validation must be formatted as an email address. This validation rule utilizes the [`egulias/email-validator`](https://github.com/egulias/EmailValidator) package for validating the email address. By default the `RFCValidation` validator is applied, but you can apply other validation styles as well:

```
    'email' => 'email:rfc,dns'
```

The example above will apply the `RFCValidation` and `DNSCheckValidation` validations. Here's a full list of validation styles you can apply:

- `rfc`: `RFCValidation`
- `strict`: `NoRFCWarningsValidation`
- `dns`: `DNSCheckValidation`
- `spoof`: `SpoofCheckValidation`
- `filter`: `FilterEmailValidation`

The `filter` validator, which uses PHP's `filter_var` function, ships with Laravel and was Laravel's default email validation behavior prior to Laravel version 5.8.

> NOTE The `dns` and `spoof` validators require the PHP `intl` extension.

<a name="rule-ends-with"></a>
ends_with:_foo_,_bar_,...

The field under validation must end with one of the given values.

<a name="rule-exclude-if"></a>
exclude_if:_anotherfield_,_value_

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the _anotherfield_ field is equal to _value_.

<a name="rule-exclude-unless"></a>
exclude_unless:_anotherfield_,_value_

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods unless _anotherfield_'s field is equal to _value_.

<a name="rule-exists"></a>
exists:_table_,_column_

The field under validation must exist in a given database table.

<a name="basic-usage-of-exists-rule"></a>
Basic Usage Of Exists Rule

```
    'state' => 'exists:states'
```

If the `column` option is not specified, the field name will be used. So, in this case, the rule will validate that the `states` database table contains a record with a `state` column value matching the request's `state` attribute value.

<a name="specifying-a-custom-column-name"></a>
Specifying A Custom Column Name

You may explicitly specify the database column name that should be used by the validation rule by placing it after the database table name:

```
    'state' => 'exists:states,abbreviation'
```

Occasionally, you may need to specify a specific database connection to be used for the `exists` query. You can accomplish this by prepending the connection name to the table name:

```
    'email' => 'exists:connection.staff,email'
```

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
    'user_id' => 'exists:App\Models\User,id'
```

If you would like to customize the query executed by the validation rule, you may use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit them:

```
    use Illuminate\Support\Facades\Validator;
    use Illuminate\Validation\Rule;
```

```
    Validator::make($data, [
        'email' => [
            'required',
            Rule::exists('staff')->where(function ($query) {
                $query->where('account_id', 1);
            }),
        ],
    ]);

<a name="rule-file"></a>
#### file

The field under validation must be a successfully uploaded file.

<a name="rule-filled"></a>
#### filled

The field under validation must not be empty when it is present.

<a name="rule-gt"></a>
#### gt:_field_

The field under validation must be greater than the given _field_. The two fields must be of the same
type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [`size`]
(#rule-size) rule.

<a name="rule-gte"></a>
#### gte:_field_

The field under validation must be greater than or equal to the given _field_. The two fields must be of
the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the
[`size`](#rule-size) rule.

<a name="rule-image"></a>
#### image

The file under validation must be an image (jpg, jpeg, png, bmp, gif, svg, or webp).

<a name="rule-in"></a>
#### in:_foo_,_bar_,...

The field under validation must be included in the given list of values. Since this rule often requires
you to `implode` an array, the `Rule::in` method may be used to fluently construct the rule:

    use Illuminate\Support\Facades\Validator;
    use Illuminate\Validation\Rule;

    Validator::make($data, [
        'zones' => [
            'required',
            Rule::in(['first-zone', 'second-zone']),
        ],
    ]);

<a name="rule-in-array"></a>
#### in_array:_anotherfield_.*

The field under validation must exist in _anotherfield_'s values.

<a name="rule-integer"></a>
#### integer

The field under validation must be an integer.

> NOTE This validation rule does not verify that the input is of the "integer" variable type, only that
the input is a string or numeric value that contains an integer.

<a name="rule-ip"></a>
#### ip

The field under validation must be an IP address.

<a name="ipv4"></a>
#### ipv4

The field under validation must be an IPv4 address.

<a name="ipv6"></a>
#### ipv6
```

The field under validation must be an IPv6 address.

<a name="rule-json"></a>
json

The field under validation must be a valid JSON string.

<a name="rule-lt"></a>
lt:_field_

The field under validation must be less than the given _field_. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [`size`](#rule-size) rule.

<a name="rule-lte"></a>
lte:_field_

The field under validation must be less than or equal to the given _field_. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [`size`](#rule-size) rule.

<a name="rule-max"></a>
max:_value_

The field under validation must be less than or equal to a maximum _value_. Strings, numerics, arrays, and files are evaluated in the same fashion as the [`size`](#rule-size) rule.

<a name="rule-mimetypes"></a>
mimetypes:_text/plain_,...

The file under validation must match one of the given MIME types:

```
    'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client's provided MIME type.

<a name="rule-mimes"></a>
mimes:_foo_,_bar_,...

The file under validation must have a MIME type corresponding to one of the listed extensions.

<a name="basic-usage-of-mime-rule"></a>
Basic Usage Of MIME Rule

```
    'photo' => 'mimes:jpg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates the MIME type of the file by reading the file's contents and guessing its MIME type. A full listing of MIME types and their corresponding extensions may be found at the following location:

[https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types]
(https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types)

<a name="rule-min"></a>
min:_value_

The field under validation must have a minimum _value_. Strings, numerics, arrays, and files are evaluated in the same fashion as the [`size`](#rule-size) rule.

<a name="multiple-of"></a>
multiple_of:_value_

The field under validation must be a multiple of _value_.

<a name="rule-not-in"></a>
not_in:_foo_,_bar_,...

The field under validation must not be included in the given list of values. The `Rule::notIn` method may be used to fluently construct the rule:

```
    use Illuminate\Validation\Rule;

    Validator::make($data, [
        'toppings' => [
            'required',
            Rule::notIn(['sprinkles', 'cherries']),
        ],
    ]);
```

```
<a name="rule-not-regex"></a>
#### not_regex:_pattern_

The field under validation must not match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same
formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' =>
'not_regex:/^.+$/i'`.

> NOTE When using the `regex` / `not_regex` patterns, it may be necessary to specify your validation
rules using an array instead of using `|` delimiters, especially if the regular expression contains a
`|` character.

<a name="rule-nullable"></a>
#### nullable

The field under validation may be `null`.

<a name="rule-numeric"></a>
#### numeric

The field under validation must be numeric.

<a name="rule-password"></a>
#### password

The field under validation must match the authenticated user's password. You may specify an
[authentication guard](/docs/{{version}}/authentication) using the rule's first parameter:

    'password' => 'password:api'

<a name="rule-present"></a>
#### present

The field under validation must be present in the input data but can be empty.

<a name="rule-regex"></a>
#### regex:_pattern_

The field under validation must match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same
formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' =>
'regex:/^.+@.+$/i'`.

> NOTE When using the `regex` / `not_regex` patterns, it may be necessary to specify rules in an array
instead of using `|` delimiters, especially if the regular expression contains a `|` character.

<a name="rule-required"></a>
#### required

The field under validation must be present in the input data and not empty. A field is considered
"empty" if one of the following conditions are true:



- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.



<a name="rule-required-if"></a>
#### required_if:_anotherfield_,_value_,...

The field under validation must be present and not empty if the _anotherfield_ field is equal to any
_value_.

If you would like to construct a more complex condition for the `required_if` rule, you may use the
`Rule::requiredIf` method. This methods accepts a boolean or a closure. When passed a closure, the
closure should return `true` or `false` to indicate if the field under validation is required:

    use Illuminate\Support\Facades\Validator;
    use Illuminate\Validation\Rule;

    Validator::make($request->all(), [
        'role_id' => Rule::requiredIf($request->user()->is_admin),
    ]);
```

```
    Validator::make($request->all(), [
        'role_id' => Rule::requiredIf(function () use ($request) {
            return $request->user()->is_admin;
        }),
    ]);
```

<a name="rule-required-unless"></a>
required_unless:_anotherfield_,_value_,...

The field under validation must be present and not empty unless the _anotherfield_ field is equal to any _value_.

<a name="rule-required-with"></a>
required_with:_foo_,_bar_,...

The field under validation must be present and not empty _only if_ any of the other specified fields are present.

<a name="rule-required-with-all"></a>
required_with_all:_foo_,_bar_,...

The field under validation must be present and not empty _only if_ all of the other specified fields are present.

<a name="rule-required-without"></a>
required_without:_foo_,_bar_,...

The field under validation must be present and not empty _only when_ any of the other specified fields are not present.

<a name="rule-required-without-all"></a>
required_without_all:_foo_,_bar_,...

The field under validation must be present and not empty _only when_ all of the other specified fields are not present.

<a name="rule-same"></a>
same:_field_

The given _field_ must match the field under validation.

<a name="rule-size"></a>
size:_value_

The field under validation must have a size matching the given _value_. For string data, _value_ corresponds to the number of characters. For numeric data, _value_ corresponds to a given integer value (the attribute must also have the `numeric` or `integer` rule). For an array, _size_ corresponds to the `count` of the array. For files, _size_ corresponds to the file size in kilobytes. Let's look at some examples:

```
    // Validate that a string is exactly 12 characters long...
    'title' => 'size:12';

    // Validate that a provided integer equals 10...
    'seats' => 'integer|size:10';

    // Validate that an array has exactly 5 elements...
    'tags' => 'array|size:5';

    // Validate that an uploaded file is exactly 512 kilobytes...
    'image' => 'file|size:512';
```

<a name="rule-starts-with"></a>
starts_with:_foo_,_bar_,...

The field under validation must start with one of the given values.

<a name="rule-string"></a>
string

The field under validation must be a string. If you would like to allow the field to also be `null`, you should assign the `nullable` rule to the field.

<a name="rule-timezone"></a>
timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

<a name="rule-unique"></a>

unique:_table_,_column_,_except_,_idColumn_

The field under validation must not exist within the given database table.

Specifying A Custom Table / Column Name:

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
    'email' => 'unique:App\Models\User,email_address'
```

The `column` option may be used to specify the field's corresponding database column. If the `column` option is not specified, the name of the field under validation will be used.

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

Specifying A Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. To accomplish this, you may prepend the connection name to the table name:

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

Forcing A Unique Rule To Ignore A Given ID:

Sometimes, you may wish to ignore a given ID during unique validation. For example, consider an "update profile" screen that includes the user's name, email address, and location. You will probably want to verify that the email address is unique. However, if the user only changes the name field and not the email field, you do not want a validation error to be thrown because the user is already the owner of the email address in question.

To instruct the validator to ignore the user's ID, we'll use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit the rules:

```
    use Illuminate\Support\Facades\Validator;
    use Illuminate\Validation\Rule;

    Validator::make($data, [
        'email' => [
            'required',
            Rule::unique('users')->ignore($user->id),
        ],
    ]);
```

> NOTE You should never pass any user controlled request input into the `ignore` method. Instead, you should only pass a system generated unique ID such as an auto-incrementing ID or UUID from an Eloquent model instance. Otherwise, your application will be vulnerable to an SQL injection attack.

Instead of passing the model key's value to the `ignore` method, you may also pass the entire model instance. Laravel will automatically extract the key from the model:

```
    Rule::unique('users')->ignore($user)
```

If your table uses a primary key column name other than `id`, you may specify the name of the column when calling the `ignore` method:

```
    Rule::unique('users')->ignore($user->id, 'user_id')
```

By default, the `unique` rule will check the uniqueness of the column matching the name of the attribute being validated. However, you may pass a different column name as the second argument to the `unique` method:

```
    Rule::unique('users', 'email_address')->ignore($user->id),
```

Adding Additional Where Clauses:

You may specify additional query conditions by customizing the query using the `where` method. For example, let's add a query condition that scopes the query to only search records that have an `account_id` column value of `1`:

```
    'email' => Rule::unique('users')->where(function ($query) {
        return $query->where('account_id', 1);
    })
```

<a name="rule-url"></a>
url

The field under validation must be a valid URL.

<a name="rule-uuid"></a>
uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

<a name="conditionally-adding-rules"></a>
Conditionally Adding Rules

<a name="skipping-validation-when-fields-have-certain-values"></a>
Skipping Validation When Fields Have Certain Values

You may occasionally wish to not validate a given field if another field has a given value. You may accomplish this using the `exclude_if` validation rule. In this example, the `appointment_date` and `doctor_name` fields will not be validated if the `has_appointment` field has a value of `false`:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
    'has_appointment' => 'required|bool',
    'appointment_date' => 'exclude_if:has_appointment,false|required|date',
    'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

Alternatively, you may use the `exclude_unless` rule to not validate a given field unless another field has a given value:

```
$validator = Validator::make($data, [
    'has_appointment' => 'required|bool',
    'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
    'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);
```

<a name="validating-when-present"></a>
Validating When Present

In some situations, you may wish to run validation checks against a field **only** if that field is present in the data being validated. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($request->all(), [
    'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

> TIP If you are attempting to validate a field that should always be present but may be empty, check out [this note on optional fields](#a-note-on-optional-fields)

<a name="complex-conditional-validation"></a>
Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your _static rules_ that never change:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game resale shop, or maybe they just enjoy collecting games. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
$v->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is a list of the rules we want to add. If the closure passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {
```

```
        return $input->games >= 100;
    });
```

> TIP The `$input` parameter passed to your closure will be an instance of `Illuminate\Support\Fluent` and may be used to access your input and files under validation.

<a name="validating-arrays"></a>
Validating Arrays

Validating array based form input fields doesn't have to be a pain. You may use "dot notation" to validate attributes within an array. For example, if the incoming HTTP request contains a `photos[profile]` field, you may validate it like so:

```
    use Illuminate\Support\Facades\Validator;

    $validator = Validator::make($request->all(), [
        'photos.profile' => 'required|image',
    ]);
```

You may also validate each element of an array. For example, to validate that each email in a given array input field is unique, you may do the following:

```
    $validator = Validator::make($request->all(), [
        'person.*.email' => 'email|unique:users',
        'person.*.first_name' => 'required_with:person.*.last_name',
    ]);
```

Likewise, you may use the `*` character when specifying [custom validation messages in your language files](#custom-messages-for-specific-attributes), making it a breeze to use a single validation message for array based fields:

```
    'custom' => [
        'person.*.email' => [
            'unique' => 'Each person must have a unique email address',
        ]
    ],
```

<a name="custom-validation-rules"></a>
Custom Validation Rules

<a name="using-rule-objects"></a>
Using Rule Objects

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using rule objects. To generate a new rule object, you may use the `make:rule` Artisan command. Let's use this command to generate a rule that verifies a string is uppercase. Laravel will place the new rule in the `app/Rules` directory. If this directory does not exist, Laravel will create it when you execute the Artisan command to create your rule:

```
    php artisan make:rule Uppercase
```

Once the rule has been created, we are ready to define its behavior. A rule object contains two methods: `passes` and `message`. The `passes` method receives the attribute value and name, and should return `true` or `false` depending on whether the attribute value is valid or not. The `message` method should return the validation error message that should be used when validation fails:

```
    <?php

    namespace App\Rules;

    use Illuminate\Contracts\Validation\Rule;

    class Uppercase implements Rule
    {
        /**
         * Determine if the validation rule passes.
         *
         * @param  string  $attribute
         * @param  mixed  $value
         * @return bool
         */
        public function passes($attribute, $value)
        {
            return strtoupper($value) === $value;
        }

        /**
         * Get the validation error message.
         *
```

```
         * @return string
         */
        public function message()
        {
            return 'The :attribute must be uppercase.';
        }
    }
```

You may call the `trans` helper from your `message` method if you would like to return an error message from your translation files:

```
    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return trans('validation.uppercase');
    }
```

Once the rule has been defined, you may attach it to a validator by passing an instance of the rule object with your other validation rules:

```
    use App\Rules\Uppercase;

    $request->validate([
        'name' => ['required', 'string', new Uppercase],
    ]);
```

<a name="using-closures"></a>
Using Closures

If you only need the functionality of a custom rule once throughout your application, you may use a closure instead of a rule object. The closure receives the attribute's name, the attribute's value, and a `$fail` callback that should be called if validation fails:

```
    use Illuminate\Support\Facades\Validator;

    $validator = Validator::make($request->all(), [
        'title' => [
            'required',
            'max:255',
            function ($attribute, $value, $fail) {
                if ($value === 'foo') {
                    $fail('The '.$attribute.' is invalid.');
                }
            },
        ],
    ]);
```

<a name="implicit-rules"></a>
Implicit Rules

By default, when an attribute being validated is not present or contains an empty string, normal validation rules, including custom rules, are not run. For example, the [`unique`](#rule-unique) rule will not be run against an empty string:

```
    use Illuminate\Support\Facades\Validator;

    $rules = ['name' => 'unique:users,name'];

    $input = ['name' => ''];

    Validator::make($input, $rules)->passes(); // true
```

For a custom rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create an "implicit" rule, implement the `Illuminate\Contracts\Validation\ImplicitRule` interface. This interface serves as a "marker interface" for the validator; therefore, it does not contain any additional methods you need to implement beyond the methods required by the typical `Rule` interface.

> NOTE An "implicit" rule only _implies_ that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

The Basics

Error Handling

- Introduction
- Configuration
- The Exception Handler
  - Reporting Exceptions
  - Ignoring Exceptions By Type
  - Rendering Exceptions
  - Reportable & Renderable Exceptions
- HTTP Exceptions
  - Custom HTTP Error Pages

Introduction

When you start a new Laravel project, error and exception handling is already configured for you. The `App\Exceptions\Handler` class is where all exceptions thrown by your application are logged and then rendered to the user. We'll dive deeper into this class throughout this documentation.

Configuration

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

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

The Exception Handler

Reporting Exceptions

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains a `register` method where you may register custom exception reporting and rendering callbacks. We'll examine each of these concepts in detail. Exception reporting is used to log exceptions or send them to an external service like Flare, Bugsnag or Sentry. By default, exceptions will be logged based on your logging configuration. However, you are free to log exceptions however you wish.

For example, if you need to report different types of exceptions in different ways, you may use the `reportable` method to register a closure that should be executed when an exception of a given type needs to be reported. Laravel will deduce what type of exception the closure reports by examining the type-hint of the closure:

```
use App\Exceptions\InvalidOrderException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->reportable(function (InvalidOrderException $e) {
        //
    });
}
```

When you register a custom exception reporting callback using the `reportable` method, Laravel will still log the exception using the default logging configuration for the application. If you wish to stop the propagation of the exception to the default logging stack, you may use the `stop` method when defining your reporting callback or

return `false` from the callback:

```
$this->reportable(function (InvalidOrderException $e) {
    //
})->stop();

$this->reportable(function (InvalidOrderException $e) {
    return false;
});
```

> TIP To customize the exception reporting for a given exception, you may also consider using reportable exceptions

Global Log Context

If available, Laravel automatically adds the current user's ID to every exception's log message as contextual data. You may define your own global contextual data by overriding the `context` method of your application's `App\Exceptions\Handler` class. This information will be included in every exception's log message written by your application:

```
/**
 * Get the default context variables for logging.
 *
 * @return array
 */
protected function context()
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

The `report` Helper

Sometimes you may need to report an exception but continue handling the current request. The `report` helper function allows you to quickly report an exception via the exception handler without rendering an error page to the user:

```
public function isValid($value)
{
    try {
        // Validate the value...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
}
```

Ignoring Exceptions By Type

When building your application, there will be some types of exceptions you simply want to ignore and never report. Your application's exception handler contains a `$dontReport` property which is initialized to an empty array. Any classes that you add to this property will never be reported; however, they may still have custom rendering logic:

```
use App\Exceptions\InvalidOrderException;

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

> TIP Behind the scenes, Laravel already ignores some types of errors for you, such as exceptions resulting from 404 HTTP "not found" errors or 419 HTTP responses generated by invalid CSRF tokens.

Rendering Exceptions

By default, the Laravel exception handler will convert exceptions into an HTTP response for you. However, you are free to register a custom rendering closure for exceptions of a given type. You may accomplish this via the `renderable` method of your exception handler.

The closure passed to the `renderable` method should return an instance of `Illuminate\Http\Response`, which may be generated via the `response` helper. Laravel will deduce what type of exception the closure renders by examining the type-hint of the closure:

```
use App\Exceptions\InvalidOrderException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->renderable(function (InvalidOrderException $e, $request) {
        return response()->view('errors.invalid-order', [], 500);
    });
}
```

Reportable & Renderable Exceptions

Instead of type-checking exceptions in the exception handler's `register` method, you may define `report` and `render` methods directly on your custom exceptions. When these methods exist, they will be automatically called by the framework:

```
<?php

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception
{
    /**
     * Report the exception.
     *
     * @return void
     */
    public function report()
    {
        //
    }

    /**
     * Render the exception into an HTTP response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function render($request)
    {
        return response(...);
    }
}
```

If your exception contains custom reporting logic that is only necessary when certain conditions are met, you may need to instruct Laravel to sometimes report the exception using the default exception handling configuration. To accomplish this, you may return `false` from the exception's `report` method:

```
/**
 * Report the exception.
 *
 * @return bool|void
 */
public function report()
{
    // Determine if the exception needs custom reporting...
```

```
    return false;
}
```

TIP You may type-hint any required dependencies of the `report` method and they will automatically be injected into the method by Laravel's service container.

HTTP Exceptions

Some exceptions describe HTTP error codes from the server. For example, this may be a "page not found" error (404), an "unauthorized error" (401) or even a developer generated 500 error. In order to generate such a response from anywhere in your application, you may use the `abort` helper:

```
abort(404);
```

Custom HTTP Error Pages

Laravel makes it easy to display custom error pages for various HTTP status codes. For example, if you wish to customize the error page for 404 HTTP status codes, create a `resources/views/errors/404.blade.php`. This file will be served on all 404 errors generated by your application. The views within this directory should be named to match the HTTP status code they correspond to. The `Symfony\Component\HttpKernel\Exception\HttpException` instance raised by the `abort` function will be passed to the view as an `$exception` variable:

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

You may publish Laravel's default error page templates using the `vendor:publish` Artisan command. Once the templates have been published, you may customize them to your liking:

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

The Basics

Logging

- Introduction
- Configuration
  - Available Channel Drivers
  - Channel Prerequisites
- Building Log Stacks
- Writing Log Messages
  - Writing To Specific Channels
- Monolog Channel Customization
  - Customizing Monolog For Channels
  - Creating Monolog Handler Channels
  - Creating Custom Channels Via Factories

Introduction

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

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

Under the hood, Laravel utilizes the Monolog library, which provides support for a variety of powerful log handlers. Laravel makes it a cinch to configure these handlers, allowing you to mix and match them to customize your application's log handling.

Configuration

All of the configuration options for your application's logging behavior is housed in the `config/logging.php` configuration file. This file allows you to configure your application's log channels, so be sure to review each of the available channels and their options. We'll review a few common options below.

By default, Laravel will use the `stack` channel when logging messages. The `stack` channel is used to aggregate multiple log channels into a single channel. For more information on building stacks, check out the documentation below.

Configuring The Channel Name

By default, Monolog is instantiated with a "channel name" that matches the current environment, such as `production` or `local`. To change this value, add a `name` option to your channel's configuration:

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

Available Channel Drivers

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

Name	Description

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

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

Channel Prerequisites

Configuring The Single and Daily Channels

The `single` and `daily` channels have three optional configuration options: `bubble`, `permission`, and `locking`.

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

Configuring The Papertrail Channel

The `papertrail` channel requires the `host` and `port` configuration options. You can obtain these values from Papertrail.

Configuring The Slack Channel

The `slack` channel requires a `url` configuration option. This URL should match a URL for an incoming webhook that you have configured for your Slack team.

By default, Slack will only receive logs at the `critical` level and above; however, you can adjust this in your `config/logging.php` configuration file by modifying the `level` configuration option within your Slack log channel's configuration array.

Building Log Stacks

As mentioned previously, the `stack` driver allows you to combine multiple channels into a single log channel for convenience. To illustrate how to use log stacks, let's take a look at an example configuration that you might see in a production application:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],

    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
```

```
        'level' => 'critical',
    ],
],
```

Let's dissect this configuration. First, notice our `stack` channel aggregates two other channels via its `channels` option: `syslog` and `slack`. So, when logging messages, both of these channels will have the opportunity to log the message. However, as we will see below, whether these channels actually log the message may be determined by the message's severity / "level".

Log Levels

Take note of the `level` configuration option present on the `syslog` and `slack` channel configurations in the example above. This option determines the minimum "level" a message must be in order to be logged by the channel. Monolog, which powers Laravel's logging services, offers all of the log levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, and **debug**.

So, imagine we log a message using the `debug` method:

```
Log::debug('An informational message.');
```

Given our configuration, the `syslog` channel will write the message to the system log; however, since the error message is not `critical` or above, it will not be sent to Slack. However, if we log an `emergency` message, it will be sent to both the system log and Slack since the `emergency` level is above our minimum level threshold for both channels:

```
Log::emergency('The system is down!');
```

Writing Log Messages

You may write information to the logs using the `Log` facade. As previously mentioned, the logger provides the eight logging levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info** and **debug**:

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

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

You may call any of these methods to log a message for the corresponding level. By default, the message will be written to the default log channel as configured by your `logging` configuration file:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Log;

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

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

```
    }
}
```

Contextual Information

An array of contextual data may also be passed to the log methods. This contextual data will be formatted and displayed with the log message:

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

Writing To Specific Channels

Sometimes you may wish to log a message to a channel other than your application's default channel. You may use the `channel` method on the `Log` facade to retrieve and log to any channel defined in your configuration file:

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

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

If you would like to create an on-demand logging stack consisting of multiple channels, you may use the `stack` method:

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

Monolog Channel Customization

Customizing Monolog For Channels

Sometimes you may need complete control over how Monolog is configured for an existing channel. For example, you may want to configure a custom Monolog `FormatterInterface` implementation for Laravel's built-in `single` channel.

To get started, define a `tap` array on the channel's configuration. The `tap` array should contain a list of classes that should have an opportunity to customize (or "tap" into) the Monolog instance after it is created. There is no conventional location where these classes should be placed, so you are free to create a directory within your application to contain these classes:

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

Once you have configured the `tap` option on your channel, you're ready to define the class that will customize your Monolog instance. This class only needs a single method: `__invoke`, which receives an `Illuminate\Log\Logger` instance. The `Illuminate\Log\Logger` instance proxies all method calls to the underlying Monolog instance:

```
<?php

namespace App\Logging;

use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    /**
     * Customize the given logger instance.
     *
     * @param  \Illuminate\Log\Logger  $logger
     * @return void
     */
    public function __invoke($logger)
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(new LineFormatter(
                '[%datetime%] %channel%.%level_name%: %message% %context% %extra%'
```

```
        }
    }
}
```

TIP All of your "tap" classes are resolved by the service container, so any constructor dependencies they require will automatically be injected.

Creating Monolog Handler Channels

Monolog has a variety of available handlers and Laravel does not include a built-in channel for each one. In some cases, you may wish to create a custom channel that is merely an instance of a specific Monolog handler that does not have a corresponding Laravel log driver. These channels can be easily created using the `monolog` driver.

When using the `monolog` driver, the `handler` configuration option is used to specify which handler will be instantiated. Optionally, any constructor parameters the handler needs may be specified using the `with` configuration option:

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

Monolog Formatters

When using the `monolog` driver, the Monolog `LineFormatter` will be used as the default formatter. However, you may customize the type of formatter passed to the handler using the `formatter` and `formatter_with` configuration options:

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

If you are using a Monolog handler that is capable of providing its own formatter, you may set the value of the `formatter` configuration option to `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

Creating Custom Channels Via Factories

If you would like to define an entirely custom channel in which you have full control over Monolog's instantiation and configuration, you may specify a `custom` driver type in your `config/logging.php` configuration file. Your configuration should include a `via` option that contains the name of the factory class which will be invoked to create the Monolog instance:

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

Once you have configured the `custom` driver channel, you're ready to define the class that will create your Monolog instance. This class only needs a single `__invoke` method which should return the Monolog logger instance. The method will receive the channels configuration array as its only argument:

```php
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Create a custom Monolog instance.
     *
     * @param  array  $config
     * @return \Monolog\Logger
     */
    public function __invoke(array $config)
    {
        return new Logger(...);
    }
}
```

Digging Deeper

Artisan Console

- Introduction
  - Tinker (REPL)
- Writing Commands
  - Generating Commands
  - Command Structure
  - Closure Commands
- Defining Input Expectations
  - Arguments
  - Options
  - Input Arrays
  - Input Descriptions
- Command I/O
  - Retrieving Input
  - Prompting For Input
  - Writing Output
- Registering Commands
- Programmatically Executing Commands
  - Calling Commands From Other Commands
- Stub Customization
- Events

Introduction

Artisan is the command line interface included with Laravel. Artisan exists at the root of your application as the `artisan` script and provides a number of helpful commands that can assist you while you build your application. To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

Every command also includes a "help" screen which displays and describes the command's available arguments and options. To view a help screen, precede the name of the command with `help`:

```
php artisan help migrate
```

Laravel Sail

If you are using Laravel Sail as your local development environment, remember to use the `sail` command line to invoke Artisan commands. Sail will execute your Artisan commands within your application's Docker containers:

```
./sail artisan list
```

Tinker (REPL)

Laravel Tinker is a powerful REPL for the Laravel framework, powered by the PsySH package.

Installation

All Laravel applications include Tinker by default. However, you may install Tinker using Composer if you have previously removed it from your application:

```
composer require laravel/tinker
```

> TIP Looking for a graphical UI for interacting with your Laravel application? Check out Tinkerwell!

Usage

Tinker allows you to interact with your entire Laravel application on the command line, including your Eloquent models, jobs, events, and more. To enter the Tinker environment, run the `tinker` Artisan command:

```
php artisan tinker
```

You can publish Tinker's configuration file using the `vendor:publish` command:

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```

> NOTE The `dispatch` helper function and `dispatch` method on the `Dispatchable` class depends on garbage collection to place the job on the queue. Therefore, when using tinker, you should use `Bus::dispatch` or `Queue::push` to dispatch jobs.

Command Allow List

Tinker utilizes an "allow" list to determine which Artisan commands are allowed to be run within its shell. By default, you may run the `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize`, and `up` commands. If you would like to allow more commands you may add them to the `commands` array in your `tinker.php` configuration file:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

Classes That Should Not Be Aliased

Typically, Tinker automatically aliases classes as you interact with them in Tinker. However, you may wish to never alias some classes. You may accomplish this by listing the classes in the `dont_alias` array of your `tinker.php` configuration file:

```
'dont_alias' => [
    App\Models\User::class,
],
```

Writing Commands

In addition to the commands provided with Artisan, you may build your own custom commands. Commands are typically stored in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be loaded by Composer.

Generating Commands

To create a new command, you may use the `make:command` Artisan command. This command will create a new command class in the `app/Console/Commands` directory. Don't worry if this directory does not exist in your application - it will be created the first time you run the `make:command` Artisan command:

```
php artisan make:command SendEmails
```

Command Structure

After generating your command, you should define appropriate values for the `signature` and `description` properties of the class. These properties will be used when displaying your command on the `list` screen. The `signature` property also allows you to define your command's input expectations. The `handle` method will be called when your command is executed. You may place your command logic in this method.

Let's take a look at an example command. Note that we are able to request any dependencies we need via the command's `handle` method. The Laravel service container will automatically inject all dependencies that are type-hinted in this method's signature:

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use App\Support\DripEmailer;
```

```
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send a marketing email to a user';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @param  \App\Support\DripEmailer  $drip
     * @return mixed
     */
    public function handle(DripEmailer $drip)
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

TIP For greater code reuse, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks. In the example above, note that we inject a service class to do the "heavy lifting" of sending the e-mails.

Closure Commands

Closure based commands provide an alternative to defining console commands as classes. In the same way that route closures are an alternative to controllers, think of command closures as an alternative to command classes. Within the `commands` method of your `app/Console/Kernel.php` file, Laravel loads the `routes/console.php` file:

```
/**
 * Register the closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}
```

Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application. Within this file, you may define all of your closure based console commands using the `Artisan::command` method. The `command` method accepts two arguments: the command signature and a closure which receives the commands arguments and options:

```
Artisan::command('mail:send {user}', function ($user) {
    $this->info("Sending email to: {$user}!");
});
```

The closure is bound to the underlying command instance, so you have full access to all of the helper methods you would typically be able to access on a full command class.

Type-Hinting Dependencies

In addition to receiving your command's arguments and options, command closures may also type-hint additional dependencies that you would like resolved out of the service container:

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

Closure Command Descriptions

When defining a closure based command, you may use the `purpose` method to add a description to the command. This description will be displayed when you run the `php artisan list` or `php artisan help` commands:

```
Artisan::command('mail:send {user}', function ($user) {
    // ...
})->purpose('Send a marketing email to a user');
```

Defining Input Expectations

When writing console commands, it is common to gather input from the user through arguments or options. Laravel makes it very convenient to define the input you expect from the user using the `signature` property on your commands. The `signature` property allows you to define the name, arguments, and options for the command in a single, expressive, route-like syntax.

Arguments

All user supplied arguments and options are wrapped in curly braces. In the following example, the command defines one required argument: `user`:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user}';
```

You may also make arguments optional or define default values for arguments:

```
// Optional argument...
mail:send {user?}

// Optional argument with default value...
mail:send {user=foo}
```

Options

Options, like arguments, are another form of user input. Options are prefixed by two hyphens (`--`) when they are provided via the command line. There are two types of options: those that receive a value and those that don't. Options that don't receive a value serve as a boolean "switch". Let's take a look at an example of this type of option:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

In this example, the `--queue` switch may be specified when calling the Artisan command. If the `--queue` switch is passed, the value of the option will be `true`. Otherwise, the value will be `false`:

```
php artisan mail:send 1 --queue
```

Options With Values

Next, let's take a look at an option that expects a value. If the user must specify a value for an option, you should suffix the option name with a `=` sign:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue=}';
```

In this example, the user may pass a value for the option like so. If the option is not specified when invoking the command, its value will be `null`:

```
php artisan mail:send 1 --queue=default
```

You may assign default values to options by specifying the default value after the option name. If no option value is passed by the user, the default value will be used:

```
mail:send {user} {--queue=default}
```

Option Shortcuts

To assign a shortcut when defining an option, you may specify it before the option name and use the `|` character as a delimiter to separate the shortcut from the full option name:

```
mail:send {user} {--Q|queue}
```

Input Arrays

If you would like to define arguments or options to expect multiple input values, you may use the `*` character. First, let's take a look at an example that specifies such an argument:

```
mail:send {user*}
```

When calling this method, the `user` arguments may be passed in order to the command line. For example, the following command will set the value of `user` to an array with `foo` and `bar` as its values:

```
php artisan mail:send foo bar
```

Option Arrays

When defining an option that expects multiple input values, each option value passed to the command should be prefixed with the option name:

```
mail:send {user} {--id=*}
```

```
php artisan mail:send --id=1 --id=2
```

Input Descriptions

You may assign descriptions to input arguments and options by separating the argument name from the description using a colon. If you need a little extra room to define your command, feel free to spread the definition across multiple lines:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send
                        {user : The ID of the user}
                        {--queue= : Whether the job should be queued}';
```

Command I/O

Retrieving Input

While your command is executing, you will likely need to access the values for the arguments and options accepted by your command. To do so, you may use the `argument` and `option` methods. If an argument or option does not exist, `null` will be returned:

```
/**
 * Execute the console command.
 *
 * @return int
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

If you need to retrieve all of the arguments as an `array`, call the `arguments` method:

```
$arguments = $this->arguments();
```

Options may be retrieved just as easily as arguments using the `option` method. To retrieve all of the options as an array, call the `options` method:

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options as an array...
$options = $this->options();
```

Prompting For Input

In addition to displaying output, you may also ask the user to provide input during the execution of your command. The `ask` method will prompt the user with the given question, accept their input, and then return the user's input back to your command:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $name = $this->ask('What is your name?');
}
```

The `secret` method is similar to `ask`, but the user's input will not be visible to them as they type in the console. This method is useful when asking for sensitive information such as passwords:

```
$password = $this->secret('What is the password?');
```

Asking For Confirmation

If you need to ask the user for a simple "yes or no" confirmation, you may use the `confirm` method. By default, this method will return `false`. However, if the user enters `y` or `yes` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    //
}
```

If necessary, you may specify that the confirmation prompt should return `true` by default by passing `true` as the second argument to the `confirm` method:

```
if ($this->confirm('Do you wish to continue?', true)) {
```

```
    //
}
```

Auto-Completion

The `anticipate` method can be used to provide auto-completion for possible choices. The user can still provide any answer, regardless of the auto-completion hints:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Alternatively, you may pass a closure as the second argument to the `anticipate` method. The closure will be called each time the user types an input character. The closure should accept a string parameter containing the user's input so far, and return an array of options for auto-completion:

```
$name = $this->anticipate('What is your address?', function ($input) {
    // Return auto-completion options...
});
```

Multiple Choice Questions

If you need to give the user a predefined set of choices when asking a question, you may use the `choice` method. You may set the array index of the default value to be returned if no option is chosen by passing the index as the third argument to the method:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex
);
```

In addition, the `choice` method accepts optional fourth and fifth arguments for determining the maximum number of attempts to select a valid response and whether multiple selections are permitted:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex,
    $maxAttempts = null,
    $allowMultipleSelections = false
);
```

Writing Output

To send output to the console, you may use the `line`, `info`, `comment`, `question` and `error` methods. Each of these methods will use appropriate ANSI colors for their purpose. For example, let's display some general information to the user. Typically, the `info` method will display in the console as green colored text:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    // ...

    $this->info('The command was successful!');
}
```

To display an error message, use the `error` method. Error message text is typically displayed in red:

```
$this->error('Something went wrong!');
```

You may use the `line` method to display plain, uncolored text:

```
$this->line('Display this on the screen');
```

You may use the `newLine` method to display a blank line:

```
// Write a single blank line...
$this->newLine();

// Write three blank lines...
$this->newLine(3);
```

Tables

The `table` method makes it easy to correctly format multiple rows / columns of data. All you need to do is provide the column names and the data for the table and Laravel will automatically calculate the appropriate width and height of the table for you:

```
use App\Models\User;

$this->table(
    ['Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

Progress Bars

For long running tasks, it can be helpful to show a progress bar that informs users how complete the task is. Using the `withProgressBar` method, Laravel will display a progress bar and advance its progress for each iteration over a given iterable value:

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function ($user) {
    $this->performTask($user);
});
```

Sometimes, you may need more manual control over how a progress bar is advanced. First, define the total number of steps the process will iterate through. Then, advance the progress bar after processing each item:

```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

TIP For more advanced options, check out the Symfony Progress Bar component documentation.

Registering Commands

All of your console commands are registered within your application's `App\Console\Kernel` class, which is your application's "console kernel". Within the `commands` method of this class, you will see a call to the kernel's `load` method. The `load` method will scan the `app/Console/Commands` directory and automatically register each command it contains with Artisan. You are even free to make additional calls to the `load` method to scan other directories for Artisan commands:

```
/**
 * Register the commands for the application.
 *
 * @return void
 */
protected function commands()
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/../Domain/Orders/Commands');

    // ...
}
```

If necessary, you may manually register commands by adding the command's class name to the `$commands` property of your `App\Console\Kernel` class. When Artisan boots, all the commands listed in this property will be resolved by the service container and registered with Artisan:

```
protected $commands = [
    Commands\SendEmails::class
];
```

Programmatically Executing Commands

Sometimes you may wish to execute an Artisan command outside of the CLI. For example, you may wish to execute an Artisan command from a route or controller. You may use the `call` method on the `Artisan` facade to accomplish this. The `call` method accepts either the command's signature name or class name as its first argument, and an array of command parameters as the second argument. The exit code will be returned:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    //
});
```

Alternatively, you may pass the entire Artisan command to the `call` method as a string:

```
Artisan::call('mail:send 1 --queue=default');
```

Passing Array Values

If your command defines an option that accepts an array, you may pass an array of values to that option:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

Passing Boolean Values

If you need to specify the value of an option that does not accept string values, such as the `--force` flag on the `migrate:refresh` command, you should pass `true` or `false` as the value of the option:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Queueing Artisan Commands

Using the `queue` method on the `Artisan` facade, you may even queue Artisan commands so they are processed in the background by your queue workers. Before using this method, make sure you have configured your queue and are running a queue listener:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    //
});
```

Using the `onConnection` and `onQueue` methods, you may specify the connection or queue the Artisan command should be dispatched to:

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Calling Commands From Other Commands

Sometimes you may wish to call other commands from an existing Artisan command. You may do so using the `call` method. This `call` method accepts the command name and an array of command arguments / options:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
}
```

If you would like to call another console command and suppress all of its output, you may use the `callSilently` method. The `callSilently` method has the same signature as the `call` method:

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Stub Customization

The Artisan console's `make` commands are used to create a variety of classes, such as controllers, jobs, migrations, and tests. These classes are generated using "stub" files that are populated with values based on your input. However, you may want to to make small changes to files generated by Artisan. To accomplish this, you may use the `stub:publish` command to publish the most common stubs to your application so that you can customize them:

```
php artisan stub:publish
```

The published stubs will be located within a `stubs` directory in the root of your application. Any changes you make to these stubs will be reflected when you generate their corresponding classes using Artisan's `make` commands.

Events

Artisan dispatches three events when running commands: `Illuminate\Console\Events\ArtisanStarting`, `Illuminate\Console\Events\CommandStarting`, and `Illuminate\Console\Events\CommandFinished`. The `ArtisanStarting` event is dispatched immediately when Artisan starts running. Next, the `CommandStarting` event is dispatched immediately before a command runs. Finally, the `CommandFinished` event is dispatched once a command finishes executing.

Digging Deeper

Broadcasting

- Introduction
- Server Side Installation
  - Configuration
  - Pusher Channels
  - Ably
  - Open Source Alternatives
- Client Side Installation
  - Pusher Channels
  - Ably
- Concept Overview
  - Using An Example Application
- Defining Broadcast Events
  - Broadcast Name
  - Broadcast Data
  - Broadcast Queue
  - Broadcast Conditions
  - Broadcasting & Database Transactions
- Authorizing Channels
  - Defining Authorization Routes
  - Defining Authorization Callbacks
  - Defining Channel Classes
- Broadcasting Events
  - Only To Others
- Receiving Broadcasts
  - Listening For Events
  - Leaving A Channel
  - Namespaces
- Presence Channels
  - Authorizing Presence Channels
  - Joining Presence Channels
  - Broadcasting To Presence Channels
- Client Events
- Notifications

Introduction

In many modern web applications, WebSockets are used to implement realtime, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a WebSocket connection to be handled by the client. WebSockets provide a more efficient alternative to continually polling your application's server for data changes that should be reflected in your UI.

For example, imagine your application is able to export a user's data to a CSV file and email it to them. However, creating this CSV file takes several minutes so you choose to create and mail the CSV within a queued job. When the CSV has been created and mailed to the user, we can use event broadcasting to dispatch a `App\Events\UserDataExported` event that is received by our application's JavaScript. Once the event is received, we can display a message to the user that their CSV has been emailed to them without them ever needing to refresh the page.

To assist you in building these types of features, Laravel makes it easy to "broadcast" your server-side Laravel events over a WebSocket connection. Broadcasting your Laravel events allows you to share the same event names and data between your server-side Laravel application and your client-side JavaScript application.

Supported Drivers

By default, Laravel includes two server-side broadcasting drivers for you to choose from: Pusher Channels and

Ably. However, community driven packages such as laravel-websockets provide additional broadcasting drivers that do not require commercial broadcasting providers.

> TIP Before diving into event broadcasting, make sure you have read Laravel's documentation on events and listeners.

Server Side Installation

To get started using Laravel's event broadcasting, we need to do some configuration within the Laravel application as well as install a few packages.

Event broadcasting is accomplished by a server-side broadcasting driver that broadcasts your Laravel events so that Laravel Echo (a JavaScript library) can receive them within the browser client. Don't worry - we'll walk through each part of the installation process step-by-step.

Configuration

All of your application's event broadcasting configuration is stored in the `config/broadcasting.php` configuration file. Laravel supports several broadcast drivers out of the box: Pusher Channels, Redis, and a `log` driver for local development and debugging. Additionally, a `null` driver is included which allows you to totally disable broadcasting during testing. A configuration example is included for each of these drivers in the `config/broadcasting.php` configuration file.

Broadcast Service Provider

Before broadcasting any events, you will first need to register the `App\Providers\BroadcastServiceProvider`. In a new Laravel applications, you only need to uncomment this provider in the `providers` array of your `config/app.php` configuration file. This `BroadcastServiceProvider` contains the code necessary to register the broadcast authorization routes and callbacks.

Queue Configuration

You will also need to configure and run a queue worker. All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected by events being broadcast.

Pusher Channels

If you plan to broadcast your events using Pusher Channels, you should install the Pusher Channels PHP SDK using the Composer package manager:

```
composer require pusher/pusher-php-server "~4.0"
```

Next, you should configure your Pusher Channels credentials in the `config/broadcasting.php` configuration file. An example Pusher Channels configuration is already included in this file, allowing you to quickly specify your key, secret, and application ID. Typically, these values should be set via the `PUSHER_APP_KEY`, `PUSHER_APP_SECRET`, and `PUSHER_APP_ID` environment variables:

```
PUSHER_APP_ID=your-pusher-app-id
PUSHER_APP_KEY=your-pusher-key
PUSHER_APP_SECRET=your-pusher-secret
PUSHER_APP_CLUSTER=mt1
```

The `config/broadcasting.php` file's `pusher` configuration also allows you to specify additional `options` that are supported by Channels, such as the cluster.

Next, you will need to change your broadcast driver to `pusher` in your `.env` file:

```
BROADCAST_DRIVER=pusher
```

Finally, you are ready to install and configure Laravel Echo, which will receive the broadcast events on the client-side.

Pusher Compatible Laravel Websockets

The laravel-websockets package is a pure PHP, Pusher compatible WebSocket package for Laravel. This package allows you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using this package, please consult its official documentation.

Ably

If you plan to broadcast your events using Ably, you should install the Ably PHP SDK using the Composer package manager:

```
composer require ably/ably-php
```

Next, you should configure your Ably credentials in the `config/broadcasting.php` configuration file. An example Ably configuration is already included in this file, allowing you to quickly specify your key. Typically, this value should be set via the `ABLY_KEY` environment variable:

```
ABLY_KEY=your-ably-key
```

Next, you will need to change your broadcast driver to `ably` in your `.env` file:

```
BROADCAST_DRIVER=ably
```

Finally, you are ready to install and configure Laravel Echo, which will receive the broadcast events on the client-side.

Open Source Alternatives

The laravel-websockets package is a pure PHP, Pusher compatible WebSocket package for Laravel. This package allows you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using this package, please consult its official documentation.

Client Side Installation

Pusher Channels

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package since we will be using the Pusher Channels broadcaster:

```
npm install --save-dev laravel-echo pusher-js
```

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file - you simply need to uncomment it:

```
import Echo from 'laravel-echo';

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run dev
```

TIP To learn more about compiling your application's JavaScript assets, please consult the documentation

on Laravel Mix.

Using An Existing Client Instance

If you already have a pre-configured Pusher Channels client instance that you would like Echo to utilize, you may pass it to Echo via the `client` configuration option:

```
import Echo from 'laravel-echo';

const client = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    client: client
});
```

Ably

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package.

You may wonder why we would install the `pusher-js` JavaScript library even though we are using Ably to broadcast our events. Thankfully, Ably includes a Pusher compatibility mode which lets us use the Pusher protocol when listening for events in our client-side application:

```
npm install --save-dev laravel-echo pusher-js
```

Before continuing, you should enable Pusher protocol support in your Ably application settings. You may enable this feature within the "Protocol Adapter Settings" portion of your Ably application's settings dashboard.

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file; however, the default configuration in the `bootstrap.js` file is intended for Pusher. You may copy the configuration below to transition your configuration to Ably:

```
import Echo from 'laravel-echo';

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_ABLY_PUBLIC_KEY,
    wsHost: 'realtime-pusher.ably.io',
    wsPort: 443,
    disableStats: true,
    encrypted: true,
});
```

Note that our Ably Echo configuration references a `MIX_ABLY_PUBLIC_KEY` environment variable. This variable's value should be your Ably public key. Your public key is the portion of your Ably key that occurs before the `:` character.

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run dev
```

> TIP To learn more about compiling your application's JavaScript assets, please consult the documentation on Laravel Mix.

Concept Overview

Laravel's event broadcasting allows you to broadcast your server-side Laravel events to your client-side JavaScript application using a driver-based approach to WebSockets. Currently, Laravel ships with Pusher Channels and Ably drivers. The events may be easily consumed on the client-side using the Laravel Echo JavaScript package.

Events are broadcast over "channels", which may be specified as public or private. Any visitor to your application may subscribe to a public channel without any authentication or authorization; however, in order to subscribe to a private channel, a user must be authenticated and authorized to listen on that channel.

> TIP If you would like to use an open source, PHP driven alternative to Pusher, check out the laravel-websockets package.

Using An Example Application

Before diving into each component of event broadcasting, let's take a high level overview using an e-commerce store as an example.

In our application, let's assume we have a page that allows users to view the shipping status for their orders. Let's also assume that a `OrderShipmentStatusUpdated` event is fired when a shipping status update is processed by the application:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

The `ShouldBroadcast` Interface

When a user is viewing one of their orders, we don't want them to have to refresh the page to view status updates. Instead, we want to broadcast the updates to the application as they are created. So, we need to mark the `OrderShipmentStatusUpdated` event with the `ShouldBroadcast` interface. This will instruct Laravel to broadcast the event when it is fired:

```
<?php

namespace App\Events;

use App\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * The order instance.
     *
     * @var \App\Order
     */
    public $order;
}
```

The `ShouldBroadcast` interface requires our event to define a `broadcastOn` method. This method is responsible for returning the channels that the event should broadcast on. An empty stub of this method is already defined on generated event classes, so we only need to fill in its details. We only want the creator of the order to be able to view status updates, so we will broadcast the event on a private channel that is tied to the order:

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return \Illuminate\Broadcasting\PrivateChannel
 */
public function broadcastOn()
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

Authorizing Channels

Remember, users must be authorized to listen on private channels. We may define our channel authorization rules in our application's `routes/channels.php` file. In this example, we need to verify that any user attempting to listen on the private `order.1` channel is actually the creator of the order:

```
use App\Models\Order;

Broadcast::channel('orders.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Listening For Event Broadcasts

Next, all that remains is to listen for the event in our JavaScript application. We can do this using Laravel Echo. First, we'll use the `private` method to subscribe to the private channel. Then, we may use the `listen` method to listen for the `OrderShipmentStatusUpdated` event. By default, all of the event's public properties will be included on the broadcast event:

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
    });
```

Defining Broadcast Events

To inform Laravel that a given event should be broadcast, you must implement the `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface on the event class. This interface is already imported into all event classes generated by the framework so you may easily add it to any of your events.

The `ShouldBroadcast` interface requires you to implement a single method: `broadcastOn`. The `broadcastOn` method should return a channel or array of channels that the event should broadcast on. The channels should be instances of `Channel`, `PrivateChannel`, or `PresenceChannel`. Instances of `Channel` represent public channels that any user may subscribe to, while `PrivateChannels` and `PresenceChannels` represent private channels that require channel authorization:

```
<?php

namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * The user that created the server.
     *
     * @var \App\Models\User
     */
    public $user;

    /**
     * Create a new event instance.
     *
```

```
     * @param  \App\Models\User  $user
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

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

After implementing the `ShouldBroadcast` interface, you only need to fire the event as you normally would. Once the event has been fired, a queued job will automatically broadcast the event using your specified broadcast driver.

Broadcast Name

By default, Laravel will broadcast the event using the event's class name. However, you may customize the broadcast name by defining a `broadcastAs` method on the event:

```
/**
 * The event's broadcast name.
 *
 * @return string
 */
public function broadcastAs()
{
    return 'server.created';
}
```

If you customize the broadcast name using the `broadcastAs` method, you should make sure to register your listener with a leading `.` character. This will instruct Echo to not prepend the application's namespace to the event:

```
.listen('.server.created', function (e) {
    ....
});
```

Broadcast Data

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

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

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

```
/**
 * Get the data to broadcast.
 *
 * @return array
 */
public function broadcastWith()
{
```

```
    return ['id' => $this->user->id];
}
```

Broadcast Queue

By default, each broadcast event is placed on the default queue for the default queue connection specified in your `queue.php` configuration file. You may customize the queue connection and name used by the broadcaster by defining `connection` and `queue` properties on your event class:

```
/**
 * The name of the queue connection to use when broadcasting the event.
 *
 * @var string
 */
public $connection = 'redis';

/**
 * The name of the queue on which to place the broadcasting job.
 *
 * @var string
 */
public $queue = 'default';
```

If you want to broadcast your event using the `sync` queue instead of the default queue driver, you can implement the `ShouldBroadcastNow` interface instead of `ShouldBroadcast`:

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class OrderShipmentStatusUpdated implements ShouldBroadcastNow
{
    //
}
```

Broadcast Conditions

Sometimes you want to broadcast your event only if a given condition is true. You may define these conditions by adding a `broadcastWhen` method to your event class:

```
/**
 * Determine if this event should broadcast.
 *
 * @return bool
 */
public function broadcastWhen()
{
    return $this->order->value > 100;
}
```

Broadcasting & Database Transactions

When broadcast events are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your event depends on these models, unexpected errors can occur when the job that broadcasts the event is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular broadcast event should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the event class:

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
```

```
{
    use SerializesModels;

    public $afterCommit = true;
}
```

TIP To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Authorizing Channels

Private channels require you to authorize that the currently authenticated user can actually listen on the channel. This is accomplished by making an HTTP request to your Laravel application with the channel name and allowing your application to determine if the user can listen on that channel. When using Laravel Echo, the HTTP request to authorize subscriptions to private channels will be made automatically; however, you do need to define the proper routes to respond to these requests.

Defining Authorization Routes

Thankfully, Laravel makes it easy to define the routes to respond to channel authorization requests. In the `App\Providers\BroadcastServiceProvider` included with your Laravel application, you will see a call to the `Broadcast::routes` method. This method will register the `/broadcasting/auth` route to handle authorization requests:

```
Broadcast::routes();
```

The `Broadcast::routes` method will automatically place its routes within the `web` middleware group; however, you may pass an array of route attributes to the method if you would like to customize the assigned attributes:

```
Broadcast::routes($attributes);
```

Customizing The Authorization Endpoint

By default, Echo will use the `/broadcasting/auth` endpoint to authorize channel access. However, you may specify your own authorization endpoint by passing the `authEndpoint` configuration option to your Echo instance:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    authEndpoint: '/custom/endpoint/auth'
});
```

Defining Authorization Callbacks

Next, we need to define the logic that will actually determine if the currently authenticated user can listen to a given channel. This is done in the `routes/channels.php` file that is included with your application. In this file, you may use the `Broadcast::channel` method to register channel authorization callbacks:

```
Broadcast::channel('orders.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Authorization Callback Model Binding

Just like HTTP routes, channel routes may also take advantage of implicit and explicit route model binding. For

example, instead of receiving a string or numeric order ID, you may request an actual `Order` model instance:

```
use App\Models\Order;

Broadcast::channel('orders.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

> NOTE Unlike HTTP route model binding, channel model binding does not support automatic implicit model binding scoping. However, this is rarely a problem because most channels can be scoped based on a single model's unique, primary key.

Authorization Callback Authentication

Private and presence broadcast channels authenticate the current user via your application's default authentication guard. If the user is not authenticated, channel authorization is automatically denied and the authorization callback is never executed. However, you may assign multiple, custom guards that should authenticate the incoming request if necessary:

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]);
```

Defining Channel Classes

If your application is consuming many different channels, your `routes/channels.php` file could become bulky. So, instead of using closures to authorize channels, you may use channel classes. To generate a channel class, use the `make:channel` Artisan command. This command will place a new channel class in the `App/Broadcasting` directory.

```
php artisan make:channel OrderChannel
```

Next, register your channel in your `routes/channels.php` file:

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

Finally, you may place the authorization logic for your channel in the channel class' `join` method. This `join` method will house the same logic you would have typically placed in your channel authorization closure. You may also take advantage of channel model binding:

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * Create a new channel instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Authenticate the user's access to the channel.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Order  $order
     * @return array|bool
     */
    public function join(User $user, Order $order)
    {
        return $user->id === $order->user_id;
```

```
    }
}
```

> TIP Like many other classes in Laravel, channel classes will automatically be resolved by the service container. So, you may type-hint any dependencies required by your channel in its constructor.

Broadcasting Events

Once you have defined an event and marked it with the `ShouldBroadcast` interface, you only need to fire the event using the event's dispatch method. The event dispatcher will notice that the event is marked with the `ShouldBroadcast` interface and will queue the event for broadcasting:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order));
```

Only To Others

When building an application that utilizes event broadcasting, you may occasionally need to broadcast an event to all subscribers to a given channel except for the current user. You may accomplish this using the `broadcast` helper and the `toOthers` method:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

To better understand when you may want to use the `toOthers` method, let's imagine a task list application where a user may create a new task by entering a task name. To create a task, your application might make a request to a `/task` URL which broadcasts the task's creation and returns a JSON representation of the new task. When your JavaScript application receives the response from the end-point, it might directly insert the new task into its task list like so:

```
axios.post('/task', task)
    .then((response) => {
        this.tasks.push(response.data);
    });
```

However, remember that we also broadcast the task's creation. If your JavaScript application is also listening for this event in order to add tasks to the task list, you will have duplicate tasks in your list: one from the end-point and one from the broadcast. You may solve this by using the `toOthers` method to instruct the broadcaster to not broadcast the event to the current user.

> NOTE Your event must use the `Illuminate\Broadcasting\InteractsWithSockets` trait in order to call the `toOthers` method.

Configuration

When you initialize a Laravel Echo instance, a socket ID is assigned to the connection. If you are using a global Axios instance to make HTTP requests from your JavaScript application, the socket ID will automatically be attached to every outgoing request as a `X-Socket-ID` header. Then, when you call the `toOthers` method, Laravel will extract the socket ID from the header and instruct the broadcaster to not broadcast to any connections with that socket ID.

If you are not using a global Axios instance, you will need to manually configure your JavaScript application to send the `X-Socket-ID` header with all outgoing requests. You may retrieve the socket ID using the `Echo.socketId` method:

```
var socketId = Echo.socketId();
```

Receiving Broadcasts

Listening For Events

Once you have installed and instantiated Laravel Echo, you are ready to start listening for events that are broadcast from your Laravel application. First, use the `channel` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event:

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
    });
```

If you would like to listen for events on a private channel, use the `private` method instead. You may continue to chain calls to the `listen` method to listen for multiple events on a single channel:

```
Echo.private(`orders.${this.order.id}`)
    .listen(...)
    .listen(...)
    .listen(...);
```

Leaving A Channel

To leave a channel, you may call the `leaveChannel` method on your Echo instance:

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

If you would like to leave a channel and also its associated private and presence channels, you may call the `leave` method:

```
Echo.leave(`orders.${this.order.id}`);
```

Namespaces

You may have noticed in the examples above that we did not specify the full `App\Events` namespace for the event classes. This is because Echo will automatically assume the events are located in the `App\Events` namespace. However, you may configure the root namespace when you instantiate Echo by passing a `namespace` configuration option:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    namespace: 'App.Other.Namespace'
});
```

Alternatively, you may prefix event classes with a `.` when subscribing to them using Echo. This will allow you to always specify the fully-qualified class name:

```
Echo.channel('orders')
    .listen('.Namespace\\Event\\Class', (e) => {
        //
    });
```

Presence Channels

Presence channels build on the security of private channels while exposing the additional feature of awareness of who is subscribed to the channel. This makes it easy to build powerful, collaborative application features such as notifying users when another user is viewing the same page or listing the inhabitants of a chat room.

Authorizing Presence Channels

All presence channels are also private channels; therefore, users must be authorized to access them. However, when defining authorization callbacks for presence channels, you will not return `true` if the user is authorized to join the channel. Instead, you should return an array of data about the user.

The data returned by the authorization callback will be made available to the presence channel event listeners in your JavaScript application. If the user is not authorized to join the presence channel, you should return `false` or `null`:

```
Broadcast::channel('chat.{roomId}', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

Joining Presence Channels

To join a presence channel, you may use Echo's `join` method. The `join` method will return a `PresenceChannel` implementation which, along with exposing the `listen` method, allows you to subscribe to the `here`, `joining`, and `leaving` events.

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        //
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    });
```

The `here` callback will be executed immediately once the channel is joined successfully, and will receive an array containing the user information for all of the other users currently subscribed to the channel. The `joining` method will be executed when a new user joins a channel, while the `leaving` method will be executed when a user leaves the channel.

Broadcasting To Presence Channels

Presence channels may receive events just like public or private channels. Using the example of a chatroom, we may want to broadcast `NewMessage` events to the room's presence channel. To do so, we'll return an instance of `PresenceChannel` from the event's `broadcastOn` method:

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return Channel|array
 */
public function broadcastOn()
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

As with other events, you may use the `broadcast` helper and the `toOthers` method to exclude the current user from receiving the broadcast:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

As typical of other types of events, you may listen for events sent to presence channels using Echo's `listen` method:

```
Echo.join(`chat.${roomId}`)
    .here(...)
    .joining(...)
    .leaving(...)
    .listen('NewMessage', (e) => {
        //
    });
```

Client Events

> TIP When using Pusher Channels, you must enable the "Client Events" option in the "App Settings" section of your application dashboard in order to send client events.

Sometimes you may wish to broadcast an event to other connected clients without hitting your Laravel application at all. This can be particularly useful for things like "typing" notifications, where you want to alert

users of your application that another user is typing a message on a given screen.

To broadcast client events, you may use Echo's `whisper` method:

```
Echo.private(`chat.${roomId}`)
    .whisper('typing', {
        name: this.user.name
    });
```

To listen for client events, you may use the `listenForWhisper` method:

```
Echo.private(`chat.${roomId}`)
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

Notifications

By pairing event broadcasting with notifications, your JavaScript application may receive new notifications as they occur without needing to refresh the page. Before getting started, be sure to read over the documentation on using the broadcast notification channel.

Once you have configured a notification to use the broadcast channel, you may listen for the broadcast events using Echo's `notification` method. Remember, the channel name should match the class name of the entity receiving the notifications:

```
Echo.private(`App.Models.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
    });
```

In this example, all notifications sent to `App\Models\User` instances via the `broadcast` channel would be received by the callback. A channel authorization callback for the `App.Models.User.{id}` channel is included in the default `BroadcastServiceProvider` that ships with the Laravel framework.

Digging Deeper

Cache

- Introduction
- Configuration
  - Driver Prerequisites
- Cache Usage
  - Obtaining A Cache Instance
  - Retrieving Items From The Cache
  - Storing Items In The Cache
  - Removing Items From The Cache
  - The Cache Helper
- Cache Tags
  - Storing Tagged Cache Items
  - Accessing Tagged Cache Items
  - Removing Tagged Cache Items
- Atomic Locks
  - Driver Prerequisites
  - Managing Locks
  - Managing Locks Across Processes
- Adding Custom Cache Drivers
  - Writing The Driver
  - Registering The Driver
- Events

Introduction

Some of the data retrieval or processing tasks performed by your application could be CPU intensive or take several seconds to complete. When this is the case, it is common to cache the retrieved data for a time so it can retrieved quickly on subsequent requests for the same data. The cached data is usually stored in a very fast data store such as Memcached or Redis.

Thankfully, Laravel provides an expressive, unified API for various cache backends, allowing you to take advantage of their blazing fast data retrieval and speed up your web application.

Configuration

Your application's cache configuration file is located at `config/cache.php`. In this file you may specify which cache driver you would like to be used by default throughout your application. Laravel supports popular caching backends like Memcached, Redis, DynamoDB, and relational databases out of the box. In addition, a file based cache driver is available, while `array` and "null" cache drivers provide convenient cache backends for your automated tests.

The cache configuration file also contains various other options, which are documented within the file, so make sure to read over these options. By default, Laravel is configured to use the `file` cache driver, which stores the serialized, cached objects on the server's filesystem. For larger applications, it is recommended that you use a more robust driver such as Memcached or Redis. You may even configure multiple cache configurations for the same driver.

Driver Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain the cache items. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache', function ($table) {
    $table->string('key')->unique();
```

```
    $table->text('value');
    $table->integer('expiration');
});
```

TIP You may also use the `php artisan cache:table` Artisan command to generate a migration with the proper schema.

Memcached

Using the Memcached driver requires the Memcached PECL package to be installed. You may list all of your Memcached servers in the `config/cache.php` configuration file. This file already contains a `memcached.servers` entry to get you started:

```
'memcached' => [
    'servers' => [
        [
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),
            'port' => env('MEMCACHED_PORT', 11211),
            'weight' => 100,
        ],
    ],
],
```

If needed, you may set the `host` option to a UNIX socket path. If you do this, the `port` option should be set to `0`:

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

Redis

Before using a Redis cache with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. Laravel Sail already includes this extension. In addition, official Laravel deployment platforms such as Laravel Forge and Laravel Vapor have the PhpRedis extension installed by default.

For more information on configuring Redis, consult its Laravel documentation page.

Cache Usage

Obtaining A Cache Instance

To obtain a cache store instance, you may use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Laravel cache contracts:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

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

        //
    }
```

```
}
```

Accessing Multiple Cache Stores

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

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

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

Retrieving Items From The Cache

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

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

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

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

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

Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache. This method will also return `false` if the item exists but its value is `null`:

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

Incrementing / Decrementing Values

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

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

Retrieve & Store

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

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

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

You may use the `rememberForever` method to retrieve an item from the cache or store it forever if it does not exist:

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

Retrieve & Delete

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

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

Storing Items In The Cache

You may use the `put` method on the `Cache` facade to store items in the cache:

```
Cache::put('key', 'value', $seconds = 10);
```

If the storage time is not passed to the `put` method, the item will be stored indefinitely:

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

Instead of passing the number of seconds as an integer, you may also pass a `DateTime` instance representing the desired expiration time of the cached item:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

Store If Not Present

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

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

Storing Items Forever

The `forever` method may be used to store an item in the cache permanently. Since these items will not expire, they must be manually removed from the cache using the `forget` method:

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

> TIP If you are using the Memcached driver, items that are stored "forever" may be removed when the cache reaches its size limit.

Removing Items From The Cache

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

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

You may also remove items by providing a zero or negative number of expiration seconds:

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

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

You may clear the entire cache using the `flush` method:

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

> NOTE Flushing the cache does not respect your configured cache "prefix" and will remove all entries from the cache. Consider this carefully when clearing a cache which is shared by other applications.

The Cache Helper

In addition to using the `Cache` facade, you may also use the global `cache` function to retrieve and store data via the cache. When the `cache` function is called with a single, string argument, it will return the value of the given key:

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

If you provide an array of key / value pairs and an expiration time to the function, it will store values in the cache for the specified duration:

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

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

When the `cache` function is called without any arguments, it returns an instance of the `Illuminate\Contracts\Cache\Factory` implementation, allowing you to call other caching methods:

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

> TIP When testing call to the global `cache` function, you may use the `Cache::shouldReceive` method just as if you were testing the facade.

Cache Tags

> NOTE Cache tags are not supported when using the `file`, `dynamodb`, or `database` cache drivers. Furthermore, when using multiple tags with caches that are stored "forever", performance will be best with a driver such as `memcached`, which automatically purges stale records.

Storing Tagged Cache Items

Cache tags allow you to tag related items in the cache and then flush all cached values that have been assigned a given tag. You may access a tagged cache by passing in an ordered array of tag names. For example, let's access a tagged cache and `put` a value into the cache:

```
Cache::tags(['people', 'artists'])->put('John', $john, $seconds);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);
```

Accessing Tagged Cache Items

To retrieve a tagged cache item, pass the same ordered list of tags to the `tags` method and then call the `get` method with the key you wish to retrieve:

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

Removing Tagged Cache Items

You may flush all items that are assigned a tag or list of tags. For example, this statement would remove all caches tagged with either `people`, `authors`, or both. So, both `Anne` and `John` would be removed from the cache:

```
Cache::tags(['people', 'authors'])->flush();
```

In contrast, this statement would remove only cached values tagged with `authors`, so `Anne` would be removed, but not `John`:

```
Cache::tags('authors')->flush();
```

Atomic Locks

> NOTE To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`,

or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Driver Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain your application's cache locks. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache_locks', function ($table) {
    $table->string('key')->primary();
    $table->string('owner');
    $table->integer('expiration');
});
```

Managing Locks

Atomic locks allow for the manipulation of distributed locks without worrying about race conditions. For example, Laravel Forge uses atomic locks to ensure that only one remote task is being executed on a server at a time. You may create and manage locks using the `Cache::lock` method:

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

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

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

    $lock->release();
}
```

The `get` method also accepts a closure. After the closure is executed, Laravel will automatically release the lock:

```
Cache::lock('foo')->get(function () {
    // Lock acquired indefinitely and automatically released...
});
```

If the lock is not available at the moment you request it, you may instruct Laravel to wait for a specified number of seconds. If the lock can not be acquired within the specified time limit, an `Illuminate\Contracts\Cache\LockTimeoutException` will be thrown:

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

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

try {
    $lock->block(5);

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

The example above may be simplified by passing a closure to the `block` method. When a closure is passed to this method, Laravel will attempt to acquire the lock for the specified number of seconds and will automatically release the lock once the closure has been executed:

```
Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting maximum of 5 seconds...
});
```

Managing Locks Across Processes

Sometimes, you may wish to acquire a lock in one process and release it in another process. For example, you

may acquire a lock during a web request and wish to release the lock at the end of a queued job that is triggered by that request. In this scenario, you should pass the lock's scoped "owner token" to the queued job so that the job can re-instantiate the lock using the given token.

In the example below, we will dispatch a queued job if a lock is successfully acquired. In addition, we will pass the lock's owner token to the queued job via the lock's `owner` method:

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

$lock = Cache::lock('processing', 120);

if ($result = $lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

Within our application's `ProcessPodcast` job, we can restore and release the lock using the owner token:

```
Cache::restoreLock('processing', $this->owner)->release();
```

If you would like to release a lock without respecting its current owner, you may use the `forceRelease` method:

```
Cache::lock('processing')->forceRelease();
```

Adding Custom Cache Drivers

Writing The Driver

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` contract. So, a MongoDB cache implementation might look something like this:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

We just need to implement each of these methods using a MongoDB connection. For an example of how to implement each of these methods, take a look at the `Illuminate\Cache\MemcachedStore` in the Laravel framework source code. Once our implementation is complete, we can finish our custom driver registration by calling the `Cache` facade's `extend` method:

```
Cache::extend('mongo', function ($app) {
    return Cache::repository(new MongoStore);
});
```

> TIP If you're wondering where to put your custom cache driver code, you could create an `Extensions` namespace within your `app` directory. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Registering The Driver

To register the custom cache driver with Laravel, we will use the `extend` method on the `Cache` facade. Since other service providers may attempt to read cached values within their `boot` method, we will register our custom driver within a `booting` callback. By using the `booting` callback, we can ensure that the custom driver is

registered just before the `boot` method is called on our application's service providers but after the `register` method is called on all of the service providers. We will register our `booting` callback within the `register` method of our application's `App\Providers\AppServiceProvider` class:

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->booting(function () {
             Cache::extend('mongo', function ($app) {
                 return Cache::repository(new MongoStore);
             });
         });
    }

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

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `config/cache.php` configuration file. The second argument is a closure that should return an `Illuminate\Cache\Repository` instance. The closure will be passed an `$app` instance, which is an instance of the service container.

Once your extension is registered, update your `config/cache.php` configuration file's `driver` option to the name of your extension.

Events

To execute code on every cache operation, you may listen for the events fired by the cache. Typically, you should place these event listeners within your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],

    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],

    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],

    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

Digging Deeper

Collections

- Introduction
  - Creating Collections
  - Extending Collections
- Available Methods
- Higher Order Messages
- Lazy Collections
  - Introduction
  - Creating Lazy Collections
  - The Enumerable Contract
  - Lazy Collection Methods

Introduction

The `Illuminate\Support\Collection` class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the `collect` helper to create a new collection instance from the array, run the `strtoupper` function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});
```

As you can see, the `Collection` class allows you to chain its methods to perform fluent mapping and reducing of the underlying array. In general, collections are immutable, meaning every `Collection` method returns an entirely new `Collection` instance.

Creating Collections

As mentioned above, the `collect` helper returns a new `Illuminate\Support\Collection` instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]);
```

TIP The results of Eloquent queries are always returned as `Collection` instances.

Extending Collections

Collections are "macroable", which allows you to add additional methods to the `Collection` class at run time. The `Illuminate\Support\Collection` class' `macro` method accepts a closure that will be executed when your macro is called. The macro closure may access the collection's other methods via `$this`, just as if it were a real method of the collection class. For example, the following code adds a `toUpper` method to the `Collection` class:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Typically, you should declare collection macros in the `boot` method of a service provider.

Macro Arguments

If necessary, you may define macros that accept additional arguments:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;
use Illuminate\Support\Str;

Collection::macro('toLocale', function ($locale) {
    return $this->map(function ($value) use ($locale) {
        return Lang::get($value, $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

Available Methods

For the majority of the remaining collection documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulate the underlying array. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary:

all average avg chunk chunkWhile collapse collect combine concat contains containsStrict count countBy crossJoin dd diff diffAssoc diffKeys dump duplicates duplicatesStrict each eachSpread every except filter first firstWhere flatMap flatten flip forget forPage get groupBy has implode intersect intersectByKeys isEmpty isNotEmpty join keyBy keys last macro make map mapInto mapSpread mapToGroups mapWithKeys max median merge mergeRecursive min mode nth only pad partition pipe pipeInto pluck pop prepend pull push put random reduce reject replace replaceRecursive reverse search shift shuffle skip skipUntil skipWhile slice some sort sortBy sortByDesc sortDesc sortKeys sortKeysDesc splice split splitIn sum take takeUntil takeWhile tap times toArray toJson transform union unique uniqueStrict unless unlessEmpty unlessNotEmpty unwrap values when whenEmpty whenNotEmpty where whereStrict whereBetween whereIn whereInStrict whereInstanceOf whereNotBetween whereNotIn whereNotInStrict whereNotNull whereNull wrap zip

Method Listing

`all()`

The `all` method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

`average()`

Alias for the `avg` method.

`avg()`

The `avg` method returns the average value of a given key:

```
$average = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->avg('foo');

// 20
```

```
$average = collect([1, 1, 2, 4])->avg();

// 2
```

chunk()

The `chunk` method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. For example, imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
```

all average avg chunk chunkWhile collapse collect combine concat contains containsStrict count countBy crossJoin dd diff diffAssoc diffKeys dump duplicates duplicatesStrict each eachSpread every except filter first firstWhere flatMap flatten flip forPage get groupBy has implode intersect intersectByKeys isEmpty isNotEmpty join keyBy keys last macro make map mapInto mapSpread mapToGroups mapWithKeys max median merge mergeRecursive min mode nth only pad partition pipe pluck random reduce reject replace replaceRecursive reverse search shuffle skip slice some sort sortBy sortByDesc sortKeys sortKeysDesc split sum take tap times toArray toJson union unique uniqueStrict unless unlessEmpty unlessNotEmpty unwrap values when whenEmpty whenNotEmpty where whereStrict whereBetween whereIn whereInStrict whereInstanceOf whereNotBetween whereNotIn whereNotInStrict wrap zip

> NOTE Methods that mutate the collection (such as `shift`, `pop`, `prepend` etc.) are **not** available on the `LazyCollection` class.

Lazy Collection Methods

In addition to the methods defined in the `Enumerable` contract, the `LazyCollection` class contains the following methods:

takeUntilTimeout()

The `takeUntilTimeout` method returns a new lazy collection that will enumerate values until the specified time. After that time, the collection will then stop enumerating:

```
$lazyCollection = LazyCollection::times(INF)
    ->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function ($number) {
    dump($number);

    sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

To illustrate the usage of this method, imagine an application that submits invoices from the database using a cursor. You could define a scheduled task that runs every 15 minutes and only processes invoices for a maximum of 14 minutes:

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
    ->takeUntilTimeout(
        Carbon::createFromTimestamp(LARAVEL_START)->add(14, 'minutes')
    )
    ->each(fn ($invoice) => $invoice->submit());
```

tapEach()

While the `each` method calls the given callback for each item in the collection right away, the `tapEach` method only calls the given callback as the items are being pulled out of the list one by one:

```
// Nothing has been dumped so far...
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {
    dump($value);
});

// Three items are dumped...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

remember()

The `remember` method returns a new lazy collection that will remember any values that have already been enumerated and will not retrieve them again on subsequent collection enumerations:

```
// No query has been executed yet...
$users = User::cursor()->remember();

// The query is executed...
// The first 5 users are hydrated from the database...
$users->take(5)->all();

// First 5 users come from the collection's cache...
// The rest are hydrated from the database...
$users->take(20)->all();
```

Digging Deeper

Compiling Assets (Mix)

- Introduction
- Installation & Setup
- Running Mix
- Working With Stylesheets
  - Tailwind CSS
  - PostCSS
  - Sass
  - URL Processing
  - Source Maps
- Working With JavaScript
  - Vue
  - React
  - Vendor Extraction
  - Custom Webpack Configuration
- Versioning / Cache Busting
- Browsersync Reloading
- Environment Variables
- Notifications

Introduction

Laravel Mix, a package developed by Laracasts creator Jeffrey Way, provides a fluent API for defining webpack build steps for your Laravel application using several common CSS and JavaScript pre-processors.

In other words, Mix makes it a cinch to compile and minify your application's CSS and JavaScript files. Through simple method chaining, you can fluently define your asset pipeline. For example:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css');
```

If you've ever been confused and overwhelmed about getting started with webpack and asset compilation, you will love Laravel Mix. However, you are not required to use it while developing your application; you are free to use any asset pipeline tool you wish, or even none at all.

> TIP If you need a head start building your application with Laravel and Tailwind CSS, check out one of our application starter kits.

Installation & Setup

Installing Node

Before running Mix, you must first ensure that Node.js and NPM are installed on your machine:

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

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

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

Installing Laravel Mix

The only remaining step is to install Laravel Mix. Within a fresh installation of Laravel, you'll find a `package.json` file in the root of your directory structure. The default `package.json` file already includes

everything you need to get started using Laravel Mix. Think of this file like your `composer.json` file, except it defines Node dependencies instead of PHP dependencies. You may install the dependencies it references by running:

```
npm install
```

Running Mix

Mix is a configuration layer on top of webpack, so to run your Mix tasks you only need to execute one of the NPM scripts that is included in the default Laravel `package.json` file. When you run the `dev` or `production` scripts, all of your application's CSS and JavaScript assets will be compiled and placed in your application's `public` directory:

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run prod
```

Watching Assets For Changes

The `npm run watch` command will continue running in your terminal and watch all relevant CSS and JavaScript files for changes. Webpack will automatically recompile your assets when it detects a change to one of these files:

```
npm run watch
```

Webpack may not be able to detect your file changes in certain local development environments. If this is the case on your system, consider using the `watch-poll` command:

```
npm run watch-poll
```

Working With Stylesheets

Your application's `webpack.mix.js` file is your entry point for all asset compilation. Think of it as a light configuration wrapper around webpack. Mix tasks can be chained together to define exactly how your assets should be compiled.

Tailwind CSS

Tailwind CSS is a modern, utility-first framework for building amazing sites without ever leaving your HTML. Let's dig into how to start using it in a Laravel project with Laravel Mix. First, we should install Tailwind using NPM and generate our Tailwind configuration file:

```
npm install

npm install -D tailwindcss@npm:@tailwindcss/postcss7-compat @tailwindcss/postcss7-compat postcss@^7

npx tailwindcss init
```

The `init` command will generate a `tailwind.config.js` file. Within this file, you may configure the paths to all of your application's templates and JavaScript so that Tailwind can tree-shake unused styles when optimizing your CSS for production:

```
purge: [
    './storage/framework/views/*.php',
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
],
```

Next, you should add each of Tailwind's "layers" to your application's `resources/css/app.css` file:

```
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;
```

Once you have configured Tailwind's layers, you are ready to update your application's `webpack.mix.js` file to compile your Tailwind powered CSS:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css', [
        require('tailwindcss'),
    ]);
```

Finally, you should reference your stylesheet in your application's primary layout template. Many applications choose to store this template at `resources/views/layouts/app.blade.php`. In addition, ensure you add the responsive viewport `meta` tag if it's not already present:

```
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link href="/css/app.css" rel="stylesheet">
</head>
```

PostCSS

PostCSS, a powerful tool for transforming your CSS, is included with Laravel Mix out of the box. By default, Mix leverages the popular Autoprefixer plugin to automatically apply all necessary CSS3 vendor prefixes. However, you're free to add any additional plugins that are appropriate for your application.

First, install the desired plugin through NPM and include it in your array of plugins when calling Mix's `postCss` method. The `postCss` method accepts the path to your CSS file as its first argument and the directory where the compiled file should be placed as its second argument:

```
mix.postCss('resources/css/app.css', 'public/css', [
    require('postcss-custom-properties')
]);
```

Or, you may execute `postCss` with no additional plugins in order to achieve simple CSS compilation and minification:

```
mix.postCss('resources/css/app.css', 'public/css');
```

Sass

The `sass` method allows you to compile Sass into CSS that can be understood by web browsers. The `sass` method accepts the path to your Sass file as its first argument and the directory where the compiled file should be placed as its second argument:

```
mix.sass('resources/sass/app.scss', 'public/css');
```

You may compile multiple Sass files into their own respective CSS files and even customize the output directory of the resulting CSS by calling the `sass` method multiple times:

```
mix.sass('resources/sass/app.sass', 'public/css')
    .sass('resources/sass/admin.sass', 'public/css/admin');
```

URL Processing

Because Laravel Mix is built on top of webpack, it's important to understand a few webpack concepts. For CSS compilation, webpack will rewrite and optimize any `url()` calls within your stylesheets. While this might initially sound strange, it's an incredibly powerful piece of functionality. Imagine that we want to compile Sass that includes a relative URL to an image:

```
.example {
    background: url('../images/example.png');
}
```

> NOTE Absolute paths for any given `url()` will be excluded from URL-rewriting. For example, `url('/images/thing.png')` or `url('http://example.com/images/thing.png')` won't be modified.

By default, Laravel Mix and webpack will find `example.png`, copy it to your `public/images` folder, and then rewrite the `url()` within your generated stylesheet. As such, your compiled CSS will be:

```
.example {
    background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);
}
```

As useful as this feature may be, it's possible that your existing folder structure is already configured in a way you like. If this is the case, you may disable `url()` rewriting like so:

```
mix.sass('resources/sass/app.scss', 'public/css').options({
    processCssUrls: false
});
```

With this addition to your `webpack.mix.js` file, Mix will no longer match any `url()` or copy assets to your public directory. In other words, the compiled CSS will look just like how you originally typed it:

```
.example {
    background: url("../images/thing.png");
}
```

Source Maps

Though disabled by default, source maps may be activated by calling the `mix.sourceMaps()` method in your `webpack.mix.js` file. Though it comes with a compile/performance cost, this will provide extra debugging information to your browser's developer tools when using compiled assets:

```
mix.js('resources/js/app.js', 'public/js')
    .sourceMaps();
```

Style Of Source Mapping

Webpack offers a variety of source mapping styles. By default, Mix's source mapping style is set to `eval-source-map`, which provides a fast rebuild time. If you want to change the mapping style, you may do so using the `sourceMaps` method:

```
let productionSourceMaps = false;

mix.js('resources/js/app.js', 'public/js')
    .sourceMaps(productionSourceMaps, 'source-map');
```

Working With JavaScript

Mix provides several features to help you work with your JavaScript files, such as compiling ECMAScript 2015, module bundling, minification, and concatenating plain JavaScript files. Even better, this all works seamlessly, without requiring an ounce of custom configuration:

```
mix.js('resources/js/app.js', 'public/js');
```

With this single line of code, you may now take advantage of:

- ES2015 syntax.
- Modules
- Minification for production environments.

Vue

Mix will automatically install the Babel plugins necessary for Vue single-file component compilation support when using the `js` method. No further configuration is required:

```
mix.js('resources/js/app.js', 'public/js');
```

Once your JavaScript has been compiled, you can reference it in your application:

```
<head>
```

```
    <!-- ... -->

    <script src="/js/app.js"></script>
</head>
```

React

Mix can automatically install the Babel plugins necessary for React support. To get started, replace your call to Mix's `js` method with a call to the `react` method:

```
mix.react('resources/js/app.jsx', 'public/js');
```

Behind the scenes, Mix will download and include the appropriate `babel-preset-react` Babel plugin. Once your JavaScript has been compiled, you can reference it in your application:

```
<head>
    <!-- ... -->

    <script src="/js/app.js"></script>
</head>
```

Vendor Extraction

One potential downside to bundling all of your application-specific JavaScript with your vendor libraries such as React and Vue is that it makes long-term caching more difficult. For example, a single update to your application code will force the browser to re-download all of your vendor libraries even if they haven't changed.

If you intend to make frequent updates to your application's JavaScript, you should consider extracting all of your vendor libraries into their own file. This way, a change to your application code will not affect the caching of your large `vendor.js` file. Mix's `extract` method makes this a breeze:

```
mix.js('resources/js/app.js', 'public/js')
    .extract(['vue'])
```

The `extract` method accepts an array of all libraries or modules that you wish to extract into a `vendor.js` file. Using the snippet above as an example, Mix will generate the following files:

- `public/js/manifest.js`: *The Webpack manifest runtime*
- `public/js/vendor.js`: *Your vendor libraries*
- `public/js/app.js`: *Your application code*

To avoid JavaScript errors, be sure to load these files in the proper order:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

Custom Webpack Configuration

Behind the scenes, Laravel Mix references a pre-configured `webpack.config.js` file to get you up and running as quickly as possible. Occasionally, you may need to manually modify this file. For example, you might have a special loader or plugin that needs to be referenced. In such instances, you have two choices:

Merging Custom Configuration

Mix provides a useful `webpackConfig` method that allows you to merge any short Webpack configuration overrides. This is a particularly appealing choice, as it doesn't require you to copy and maintain your own copy of the `webpack.config.js` file. The `webpackConfig` method accepts an object, which should contain any Webpack-specific configuration that you wish to apply.

```
mix.webpackConfig({
    resolve: {
        modules: [
            path.resolve(__dirname, 'vendor/laravel/spark/resources/assets/js')
```

```
        ]
    }
});
```

Custom Configuration Files

If you would like to completely customize your Webpack configuration, copy the `node_modules/laravel-mix/setup/webpack.config.js` file to your project's root directory. Next, point all of the `--config` references in your `package.json` file to the newly copied configuration file. If you choose to take this approach to customization, any future upstream updates to Mix's `webpack.config.js` must be manually merged into your customized file.

Versioning / Cache Busting

Many developers suffix their compiled assets with a timestamp or unique token to force browsers to load the fresh assets instead of serving stale copies of the code. Mix can automatically handle this for you using the `version` method.

The `version` method will append a unique hash to the filenames of all compiled files, allowing for more convenient cache busting:

```
mix.js('resources/js/app.js', 'public/js')
    .version();
```

After generating the versioned file, you won't know the exact filename. So, you should use Laravel's global `mix` function within your views to load the appropriately hashed asset. The `mix` function will automatically determine the current name of the hashed file:

```
<script src="{{ mix('/js/app.js') }}"></script>
```

Because versioned files are usually unnecessary in development, you may instruct the versioning process to only run during `npm run prod`:

```
mix.js('resources/js/app.js', 'public/js');

if (mix.inProduction()) {
    mix.version();
}
```

Custom Mix Base URLs

If your Mix compiled assets are deployed to a CDN separate from your application, you will need to change the base URL generated by the `mix` function. You may do so by adding a `mix_url` configuration option to your application's `config/app.php` configuration file:

```
'mix_url' => env('MIX_ASSET_URL', null)
```

After configuring the Mix URL, The `mix` function will prefix the configured URL when generating URLs to assets:

```
https://cdn.example.com/js/app.js?id=1964becbdd96414518cd
```

Browsersync Reloading

BrowserSync can automatically monitor your files for changes, and inject your changes into the browser without requiring a manual refresh. You may enable support for this by calling the `mix.browserSync()` method:

```
mix.browserSync('laravel.test');
```

BrowserSync options may be specified by passing a JavaScript object to the `browserSync` method:

```
mix.browserSync({
    proxy: 'laravel.test'
});
```

Next, start webpack's development server using the `npm run watch` command. Now, when you modify a script or PHP file you can watch as the browser instantly refreshes the page to reflect your changes.

Environment Variables

You may inject environment variables into your `webpack.mix.js` script by prefixing one of the environment variables in your `.env` file with `MIX_`:

```
MIX_SENTRY_DSN_PUBLIC=http://example.com
```

After the variable has been defined in your `.env` file, you may access it via the `process.env` object. However, you will need to restart the task if the environment variable's value changes while the task is running:

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

Notifications

When available, Mix will automatically display OS notifications when compiling, giving you instant feedback as to whether the compilation was successful or not. However, there may be instances when you would prefer to disable these notifications. One such example might be triggering Mix on your production server. Notifications may be deactivated using the `disableNotifications` method:

```
mix.disableNotifications();
```

Digging Deeper

Contracts

- Introduction
  - Contracts Vs. Facades
- When To Use Contracts
- How To Use Contracts
- Contract Reference

Introduction

Laravel's "contracts" are a set of interfaces that define the core services provided by the framework. For example, an `Illuminate\Contracts\Queue\Queue` contract defines the methods needed for queueing jobs, while the `Illuminate\Contracts\Mail\Mailer` contract defines the methods needed for sending e-mail.

Each contract has a corresponding implementation provided by the framework. For example, Laravel provides a queue implementation with a variety of drivers, and a mailer implementation that is powered by SwiftMailer.

All of the Laravel contracts live in their own GitHub repository. This provides a quick reference point for all available contracts, as well as a single, decoupled package that may be utilized when building packages that interact with Laravel services.

Contracts Vs. Facades

Laravel's facades and helper functions provide a simple way of utilizing Laravel's services without needing to type-hint and resolve contracts out of the service container. In most cases, each facade has an equivalent contract.

Unlike facades, which do not require you to require them in your class' constructor, contracts allow you to define explicit dependencies for your classes. Some developers prefer to explicitly define their dependencies in this way and therefore prefer to use contracts, while other developers enjoy the convenience of facades. **In general, most applications can use facades without issue during development.**

When To Use Contracts

The decision to use contracts or facades will come down to personal taste and the tastes of your development team. Both contracts and facades can be used to create robust, well-tested Laravel applications. Contracts and facades are not mutually exclusive. Some parts of your applications may use facades while others depend on contracts. As long as you are keeping your class' responsibilities focused, you will notice very few practical differences between using contracts and facades.

In general, most applications can use facades without issue during development. If you are building a package that integrates with multiple PHP frameworks you may wish to use the `illuminate/contracts` package to define your integration with Laravel's services without the need to require Laravel's concrete implementations in your package's `composer.json` file.

How To Use Contracts

So, how do you get an implementation of a contract? It's actually quite simple.

Many types of classes in Laravel are resolved through the service container, including controllers, event listeners, middleware, queued jobs, and even route closures. So, to get an implementation of a contract, you can just "type-hint" the interface in the constructor of the class being resolved.

For example, take a look at this event listener:

```
<?php
```

```
namespace App\Listeners;

use App\Events\OrderWasPlaced;
use App\Models\User;
use Illuminate\Contracts\Redis\Factory;

class CacheOrderInformation
{
    /**
     * The Redis factory implementation.
     *
     * @var \Illuminate\Contracts\Redis\Factory
     */
    protected $redis;

    /**
     * Create a new event handler instance.
     *
     * @param  \Illuminate\Contracts\Redis\Factory  $redis
     * @return void
     */
    public function __construct(Factory $redis)
    {
        $this->redis = $redis;
    }

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderWasPlaced  $event
     * @return void
     */
    public function handle(OrderWasPlaced $event)
    {
        //
    }
}
```

When the event listener is resolved, the service container will read the type-hints on the constructor of the class, and inject the appropriate value. To learn more about registering things in the service container, check out its documentation.

Contract Reference

This table provides a quick reference to all of the Laravel contracts and their equivalent facades:

Contract	References Facade
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	`Gate`
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	`Auth`
Illuminate\Contracts\Auth\Guard	`Auth::guard()`
Illuminate\Contracts\Auth\PasswordBroker	`Password::broker()`
Illuminate\Contracts\Auth\PasswordBrokerFactory	`Password`
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	
Illuminate\Contracts\Auth\UserProvider	
Illuminate\Contracts\Bus\Dispatcher	`Bus`
Illuminate\Contracts\Bus\QueueingDispatcher	`Bus::dispatchToQueue()`
Illuminate\Contracts\Broadcasting\Factory	`Broadcast`
Illuminate\Contracts\Broadcasting\Broadcaster	`Broadcast::connection()`
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Cache\Factory	`Cache`
Illuminate\Contracts\Cache\Lock	

Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	`Cache::driver()`
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	`Config`
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	`Artisan`
Illuminate\Contracts\Container\Container	`App`
Illuminate\Contracts\Cookie\Factory	`Cookie`
Illuminate\Contracts\Cookie\QueueingFactory	`Cookie::queue()`
Illuminate\Contracts\Database\ModelIdentifier	
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	`Crypt`
Illuminate\Contracts\Events\Dispatcher	`Event`
Illuminate\Contracts\Filesystem\Cloud	`Storage::cloud()`
Illuminate\Contracts\Filesystem\Factory	`Storage`
Illuminate\Contracts\Filesystem\Filesystem	`Storage::disk()`
Illuminate\Contracts\Foundation\Application	`App`
Illuminate\Contracts\Hashing\Hasher	`Hash`
Illuminate\Contracts\Http\Kernel	
Illuminate\Contracts\Mail\MailQueue	`Mail::queue()`
Illuminate\Contracts\Mail\Mailable	
Illuminate\Contracts\Mail\Mailer	`Mail`
Illuminate\Contracts\Notifications\Dispatcher	`Notification`
Illuminate\Contracts\Notifications\Factory	`Notification`
Illuminate\Contracts\Pagination\LengthAwarePaginator	
Illuminate\Contracts\Pagination\Paginator	
Illuminate\Contracts\Pipeline\Hub	
Illuminate\Contracts\Pipeline\Pipeline	
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	
Illuminate\Contracts\Translation\Loader	
Illuminate\Contracts\Translation\Translator	`Lang`
Illuminate\Contracts\Validation\Factory	`Validator`
Illuminate\Contracts\Validation\ImplicitRule	

Illuminate\Contracts\Validation\Rule	
Illuminate\Contracts\Validation\ValidatesWhenResolved	
Illuminate\Contracts\Validation\Validator	`Validator::make()`
Illuminate\Contracts\View\Engine	
Illuminate\Contracts\View\Factory	`View`
Illuminate\Contracts\View\View	`View::make()`

Digging Deeper

Events

- Introduction
- Registering Events & Listeners
  - Generating Events & Listeners
  - Manually Registering Events
  - Event Discovery
- Defining Events
- Defining Listeners
- Queued Event Listeners
  - Manually Interacting The Queue
  - Queued Event Listeners & Database Transactions
  - Handling Failed Jobs
- Dispatching Events
- Event Subscribers
  - Writing Event Subscribers
  - Registering Event Subscribers

Introduction

Laravel's events provide a simple observer pattern implementation, allowing you to subscribe and listen for various events that occur within your application. Event classes are typically stored in the `app/Events` directory, while their listeners are stored in `app/Listeners`. Don't worry if you don't see these directories in your application as they will be created for you as you generate events and listeners using Artisan console commands.

Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other. For example, you may wish to send a Slack notification to your user each time an order has shipped. Instead of coupling your order processing code to your Slack notification code, you can raise an `App\Events\OrderShipped` event which a listener can receive and use to dispatch a Slack notification.

Registering Events & Listeners

The `App\Providers\EventServiceProvider` included with your Laravel application provides a convenient place to register all of your application's event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). You may add as many events to this array as your application requires. For example, let's add an `OrderShipped` event:

```
use App\Events\OrderShipped;
use App\Listeners\SendShipmentNotification;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    OrderShipped::class => [
        SendShipmentNotification::class,
    ],
];
```

> TIP The `event:list` command may be used to display a list of all events and listeners registered by your application.

Generating Events & Listeners

Of course, manually creating the files for each event and listener is cumbersome. Instead, add listeners and events to your `EventServiceProvider` and use the `event:generate` Artisan command. This command will generate

any events or listeners that are listed in your `EventServiceProvider` that do not already exist:

```
php artisan event:generate
```

Alternatively, you may use the `make:event` and `make:listener` Artisan commands to generate individual events and listeners:

```
php artisan make:event PodcastProcessed
```

```
php artisan make:listener SendPodcastNotification --event=PodcastProcessed
```

Manually Registering Events

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register class or closure based event listeners manually in the `boot` method of your `EventServiceProvider`:

```
use App\Events\PodcastProcessed;
use App\Listeners\SendPodcastNotification;
use Illuminate\Support\Facades\Event;

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    Event::listen(
        PodcastProcessed::class,
        [SendPodcastNotification::class, 'handle']
    );

    Event::listen(function (PodcastProcessed $event) {
        //
    });
}
```

Queueable Anonymous Event Listeners

When registering closure based event listeners manually, you may wrap the listener closure within the `Illuminate\Events\queueable` function to instruct Laravel to execute the listener using the queue:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        //
    }));
}
```

Like queued jobs, you may use the `onConnection`, `onQueue`, and `delay` methods to customize the execution of the queued listener:

```
Event::listen(queueable(function (PodcastProcessed $event) {
    //
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10)));
```

If you would like to handle anonymous queued listener failures, you may provide a closure to the `catch` method while defining the `queueable` listener. This closure will receive the event instance and the `Throwable` instance that caused the listener's failure:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
```

```
use Throwable;

Event::listen(queueable(function (PodcastProcessed $event) {
    //
})->catch(function (PodcastProcessed $event, Throwable $e) {
    // The queued listener failed...
}));
```

Wildcard Event Listeners

You may even register listeners using the `*` as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the event name as their first argument and the entire event data array as their second argument:

```
Event::listen('event.*', function ($eventName, array $data) {
    //
});
```

Event Discovery

Instead of registering events and listeners manually in the `$listen` array of the `EventServiceProvider`, you can enable automatic event discovery. When event discovery is enabled, Laravel will automatically find and register your events and listeners by scanning your application's `Listeners` directory. In addition, any explicitly defined events listed in the `EventServiceProvider` will still be registered.

Laravel finds event listeners by scanning the listener classes using PHP's reflection services. When Laravel finds any listener class method that begins with `handle`, Laravel will register those methods as event listeners for the event that is type-hinted in the method's signature:

```
use App\Events\PodcastProcessed;

class SendPodcastNotification
{
    /**
     * Handle the given event.
     *
     * @param  \App\Events\PodcastProcessed
     * @return void
     */
    public function handle(PodcastProcessed $event)
    {
        //
    }
}
```

Event discovery is disabled by default, but you can enable it by overriding the `shouldDiscoverEvents` method of your application's `EventServiceProvider`:

```
/**
 * Determine if events and listeners should be automatically discovered.
 *
 * @return bool
 */
public function shouldDiscoverEvents()
{
    return true;
}
```

By default, all listeners within your application's `app/Listeners` directory will be scanned. If you would like to define additional directories to scan, you may override the `discoverEventsWithin` method in your `EventServiceProvider`:

```
/**
 * Get the listener directories that should be used to discover events.
 *
 * @return array
 */
protected function discoverEventsWithin()
{
    return [
        $this->app->path('Listeners'),
    ];
```

```
}
```

Event Discovery In Production

In production, it is not efficient for the framework to scan all of your listeners on every request. Therefore, during your deployment process, you should run the `event:cache` Artisan command to cache a manifest of all of your application's events and listeners. This manifest will be used by the framework to speed up the event registration process. The `event:clear` command may be used to destroy the cache.

Defining Events

An event class is essentially a data container which holds the information related to the event. For example, let's assume an `App\Events\OrderShipped` event receives an Eloquent ORM object:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;

    /**
     * Create a new event instance.
     *
     * @param  \App\Models\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}
```

As you can see, this event class contains no logic. It is a container for the `App\Models\Order` instance that was purchased. The `SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's `serialize` function, such as when utilizing queued listeners.

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive event instances in their `handle` method. The `event:generate` and `make:listener` Artisan commands will automatically import the proper event class and type-hint the event on the `handle` method. Within the `handle` method, you may perform any actions necessary to respond to the event:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

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

```
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // Access the order using $event->order...
    }
}
```

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

Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so by returning `false` from your listener's `handle` method.

Queued Event Listeners

Queueing listeners can be beneficial if your listener is going to perform a slow task such as sending an email or making an HTTP request. Before using queued listeners, make sure to configure your queue and start a queue worker on your server or local development environment.

To specify that a listener should be queued, add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` and `make:listener` Artisan commands already have this interface imported into the current namespace so you can use it immediately:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}
```

That's it! Now, when an event handled by this listener is dispatched, the listener will automatically be queued by the event dispatcher using Laravel's queue system. If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has finished processing.

Customizing The Queue Connection & Queue Name

If you would like to customize the queue connection, queue name, or queue delay time of an event listener, you may define the `$connection`, `$queue`, or `$delay` properties on your listener class:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * The name of the connection the job should be sent to.
     *
     * @var string|null
     */
    public $connection = 'sqs';
```

```
    /**
     * The name of the queue the job should be sent to.
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * The time (seconds) before the job should be processed.
     *
     * @var int
     */
    public $delay = 60;
}
```

If you would like to define the listener's queue at runtime, you may define a `viaQueue` method on the listener:

```
/**
 * Get the name of the listener's queue.
 *
 * @return string
 */
public function viaQueue()
{
    return 'listeners';
}
```

Conditionally Queueing Listeners

Sometimes, you may need to determine whether a listener should be queued based on some data that's only available at runtime. To accomplish this, a `shouldQueue` method may be added to a listener to determine whether the listener should be queued. If the `shouldQueue` method returns `false`, the listener will not be executed:

```
<?php

namespace App\Listeners;

use App\Events\OrderCreated;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * Reward a gift card to the customer.
     *
     * @param  \App\Events\OrderCreated  $event
     * @return void
     */
    public function handle(OrderCreated $event)
    {
        //
    }

    /**
     * Determine whether the listener should be queued.
     *
     * @param  \App\Events\OrderCreated  $event
     * @return bool
     */
    public function shouldQueue(OrderCreated $event)
    {
        return $event->order->subtotal >= 5000;
    }
}
```

Manually Interacting The Queue

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

```
<?php

namespace App\Listeners;
```

```
use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

Queued Event Listeners & Database Transactions

When queued listeners are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your listener depends on these models, unexpected errors can occur when the job that dispatches the queued listener is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued listener should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the listener class:

```
<?php

namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public $afterCommit = true;
}
```

> TIP To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Handling Failed Jobs

Sometimes your queued event listeners may fail. If queued listener exceeds the maximum number of attempts as defined by your queue worker, the `failed` method will be called on your listener. The `failed` method receives the event instance and the `Throwable` that caused the failure:

```
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
```

```
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        //
    }

    /**
     * Handle a job failure.
     *
     * @param  \App\Events\OrderShipped  $event
     * @param  \Throwable  $exception
     * @return void
     */
    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

Specifying Queued Listener Maximum Attempts

If one of your queued listener is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a listener may be attempted.

You may define `$tries` property on your listener class to specify how many times the listener may be attempted before it is considered to have failed:

```
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * The number of times the queued listener may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

As an alternative to defining how many times a listener may be attempted before it fails, you may define a time at which the listener should no longer be attempted. This allows a listener to be attempted any number of times within a given time frame. To define the time at which a listener should no longer be attempted, add a `retryUntil` method to your listener class. This method should return a `DateTime` instance:

```
/**
 * Determine the time at which the listener should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addMinutes(5);
}
```

Dispatching Events

To dispatch an event, you may call the static `dispatch` method on the event. This method is made available on the event by the `Illuminate\Foundation\Events\Dispatchable` trait. Any arguments passed to the `dispatch` method will be passed to the event's constructor:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\Request;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $order = Order::findOrFail($request->order_id);

        // Order shipment logic...

        OrderShipped::dispatch($order);
    }
}
```

TIP When testing, it can be helpful to assert that certain events were dispatched without actually triggering their listeners. Laravel's built-in testing helpers makes it a cinch.

Event Subscribers

Writing Event Subscribers

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

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin($event) {}

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

    /**
     * Register the listeners for the subscriber.
     *
     * @param  \Illuminate\Events\Dispatcher  $events
     * @return void
     */
    public function subscribe($events)
    {
        $events->listen(
            'Illuminate\Auth\Events\Login',
            [UserEventSubscriber::class, 'handleUserLogin']
        );

        $events->listen(
            'Illuminate\Auth\Events\Logout',
            [UserEventSubscriber::class, 'handleUserLogout']
        );
    }
}
```

Registering Event Subscribers

After writing the subscriber, you are ready to register it with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventSubscriber` to the list:

```
<?php

namespace App\Providers;

use App\Listeners\UserEventSubscriber;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

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

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        UserEventSubscriber::class,
    ];
}
```

Digging Deeper

File Storage

- Introduction
- Configuration
  - The Local Driver
  - The Public Disk
  - Driver Prerequisites
  - Caching
- Obtaining Disk Instances
- Retrieving Files
  - Downloading Files
  - File URLs
  - File Metadata
- Storing Files
  - File Uploads
  - File Visibility
- Deleting Files
- Directories
- Custom Filesystems

Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3. Even better, it's amazingly simple to switch between these storage options between your local development machine and production server as the API remains the same for each system.

Configuration

Laravel's filesystem configuration file is located at `config/filesystems.php`. Within this file you may configure all of your filesystem "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file so you can modify the configuration to reflect your storage preferences and credentials.

The `local` driver interacts with files stored locally on the server running the Laravel application while the `s3` driver is used to write to Amazon's S3 cloud storage service.

> TIP You may configure as many disks as you like and may even have multiple disks that use the same driver.

The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory defined in your `filesystems` configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would write to `storage/app/example.txt`:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('local')->put('example.txt', 'Contents');
```

The Public Disk

The `public` disk included in your application's `filesystems` configuration file is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores its files in `storage/app/public`.

To make these files accessible from the web, you should create a symbolic link from `public/storage` to

`storage/app/public`. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer.

To create the symbolic link, you may use the `storage:link` Artisan command:

```
php artisan storage:link
```

Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the `asset` helper:

```
echo asset('storage/file.txt');
```

You may configure additional symbolic links in your `filesystems` configuration file. Each of the configured links will be created when you run the `storage:link` command:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

Driver Prerequisites

Composer Packages

Before using the S3 or SFTP drivers, you will need to install the appropriate package via the Composer package manager:

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- SFTP: `league/flysystem-sftp ~1.0`

In addition, you may choose to install a cached adapter for increased performance:

- CachedAdapter: `league/flysystem-cached-adapter ~1.0`

S3 Driver Configuration

The S3 driver configuration information is located in your `config/filesystems.php` configuration file. This file contains an example configuration array for an S3 driver. You are free to modify this array with your own S3 configuration and credentials. For convenience, these environment variables match the naming convention used by the AWS CLI.

FTP Driver Configuration

Laravel's Flysystem integrations works great with FTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure a FTP filesystem, you may use the configuration example below:

```
'ftp' => [
    'driver' => 'ftp',
    'host' => 'ftp.example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Optional FTP Settings...
    // 'port' => 21,
    // 'root' => '',
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

SFTP Driver Configuration

Laravel's Flysystem integrations works great with SFTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure a SFTP filesystem, you

may use the configuration example below:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => 'example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Settings for SSH key based authentication...
    'privateKey' => '/path/to/privateKey',
    'password' => 'encryption-password',

    // Optional SFTP Settings...
    // 'port' => 22,
    // 'root' => '',
    // 'timeout' => 30,
],
```

Caching

To enable caching for a given disk, you may add a `cache` directive to the disk's configuration options. The `cache` option should be an array of caching options containing the `disk` name, the `expire` time in seconds, and the cache `prefix`:

```
's3' => [
    'driver' => 's3',

    // Other Disk Options...

    'cache' => [
        'store' => 'memcached',
        'expire' => 600,
        'prefix' => 'cache-prefix',
    ],
],
```

Obtaining Disk Instances

The `Storage` facade may be used to interact with any of your configured disks. For example, you may use the `put` method on the facade to store an avatar on the default disk. If you call methods on the `Storage` facade without first calling the `disk` method, the method will automatically be passed to the default disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $content);
```

If your application interacts with multiple disks, you may use the `disk` method on the `Storage` facade to work with files on a particular disk:

```
Storage::disk('s3')->put('avatars/1', $content);
```

Retrieving Files

The `get` method may be used to retrieve the contents of a file. The raw string contents of the file will be returned by the method. Remember, all file paths should be specified relative to the disk's "root" location:

```
$contents = Storage::get('file.jpg');
```

The `exists` method may be used to determine if a file exists on the disk:

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

The `missing` method may be used to determine if a file is missing from the disk:

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

Downloading Files

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

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

File URLs

You may use the `url` method to get the URL for a given file. If you are using the `local` driver, this will typically just prepend `/storage` to the given path and return a relative URL to the file. If you are using the `s3` driver, the fully qualified remote URL will be returned:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::url('file.jpg');
```

When using the `local` driver, all files that should be publicly accessible should be placed in the `storage/app/public` directory. Furthermore, you should create a symbolic link at `public/storage` which points to the `storage/app/public` directory.

> NOTE When using the `local` driver, the return value of `url` is not URL encoded. For this reason, we recommend always storing your files using names that will create valid URLs.

Temporary URLs

Using the `temporaryUrl` method, you may create temporary URLs to files stored using the `s3` driver. This method accepts a path and a `DateTime` instance specifying when the URL should expire:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

If you need to specify additional S3 request parameters, you may pass the array of request parameters as the third argument to the `temporaryUrl` method:

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
    ]
);
```

URL Host Customization

If you would like to pre-define the host for URLs generated using the `Storage` facade, you may add a `url` option to the disk's configuration array:

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

File Metadata

In addition to reading and writing files, Laravel can also provide information about the files themselves. For

example, the `size` method may be used to get the size of a file in bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file.jpg');
```

File Paths

You may use the `path` method to get the path for a given file. If you are using the `local` driver, this will return the absolute path to the file. If you are using the `s3` driver, this method will return the relative path to the file in the S3 bucket:

```
use Illuminate\Support\Facades\Storage;

$path = Storage::path('file.jpg');
```

Storing Files

The `put` method may be used to store file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Remember, all file paths should be specified relative to the "root" location configured for the disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Automatic Streaming

Streaming files to storage offers significantly reduced memory usage. If you would like Laravel to automatically manage streaming a given file to your storage location, you may use the `putFile` or `putFileAs` method. This method accepts either an `Illuminate\Http\File` or `Illuminate\Http\UploadedFile` instance and will automatically stream the file to your desired location:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// Automatically generate a unique ID for filename...
$path = Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a filename...
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

There are a few important things to note about the `putFile` method. Note that we only specified a directory name and not a filename. By default, the `putFile` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `putFile` method so you can store the path, including the generated filename, in your database.

The `putFile` and `putFileAs` methods also accept an argument to specify the "visibility" of the stored file. This is particularly useful if you are storing the file on a cloud disk such as Amazon S3 and would like the file to be publicly accessible via generated URLs:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

Prepending & Appending To Files

The `prepend` and `append` methods allow you to write to the beginning or end of a file:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Copying & Moving Files

The `copy` method may be used to copy an existing file to a new location on the disk, while the `move` method may be used to rename or move an existing file to a new location:

```
Storage::copy('old/file.jpg', 'new/file.jpg');

Storage::move('old/file.jpg', 'new/file.jpg');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as photos and documents. Laravel makes it very easy to store uploaded files using the `store` method on an uploaded file instance. Call the `store` method with the path at which you wish to store the uploaded file:

```
<?php

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

There are a few important things to note about this example. Note that we only specified a directory name, not a filename. By default, the `store` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `store` method so you can store the path, including the generated filename, in your database.

You may also call the `putFile` method on the `Storage` facade to perform the same file storage operation as the example above:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying A File Name

If you do not want a filename to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the filename, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

You may also use the `putFileAs` method on the `Storage` facade, which will perform the same file storage operation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

> NOTE Unprintable and invalid unicode characters will automatically be removed from file paths. Therefore, you may wish to sanitize your file paths before passing them to Laravel's file storage methods. File paths are normalized using the `League\Flysystem\Util::normalizePath` method.

Specifying A Disk

By default, this uploaded file's `store` method will use your default disk. If you would like to specify another disk, pass the disk name as the second argument to the `store` method:

```
$path = $request->file('avatar')->store(
    'avatars/'.$request->user()->id, 's3'
);
```

If you are using the `storeAs` method, you may pass the disk name as the third argument to the method:

```
$path = $request->file('avatar')->storeAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Other Uploaded File Information

If you would like to get original name of the uploaded file, you may do so using the `getClientOriginalName` method:

```
$name = $request->file('avatar')->getClientOriginalName();
```

The `extension` method may be used to get the file extension of the uploaded file:

```
$extension = $request->file('avatar')->extension();
```

File Visibility

In Laravel's Flysystem integration, "visibility" is an abstraction of file permissions across multiple platforms. Files may either be declared `public` or `private`. When a file is declared `public`, you are indicating that the file should generally be accessible to others. For example, when using the S3 driver, you may retrieve URLs for `public` files.

You can set the visibility when writing the file via the `put` method:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

If the file has already been stored, its visibility can be retrieved and set via the `getVisibility` and `setVisibility` methods:

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

When interacting with uploaded files, you may use the `storePublicly` and `storePubliclyAs` methods to store the uploaded file with `public` visibility:

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Local Files & Visibility

When using the `local` driver, `public` visibility translates to `0755` permissions for directories and `0644` permissions for files. You can modify the permissions mappings in your application's `filesystems` configuration file:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0664,
            'private' => 0600,
```

```
        ],
        'dir' => [
            'public' => 0775,
            'private' => 0700,
        ],
    ],
],
```

Deleting Files

The `delete` method accepts a single filename or an array of files to delete:

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file.jpg', 'file2.jpg']);
```

If necessary, you may specify the disk that the file should be deleted from:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('path/file.jpg');
```

Directories

Get All Files Within A Directory

The `files` method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all subdirectories, you may use the `allFiles` method:

```
use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);

$files = Storage::allFiles($directory);
```

Get All Directories Within A Directory

The `directories` method returns an array of all the directories within a given directory. Additionally, you may use the `allDirectories` method to get a list of all directories within a given directory and all of its subdirectories:

```
$directories = Storage::directories($directory);

$directories = Storage::allDirectories($directory);
```

Create A Directory

The `makeDirectory` method will create the given directory, including any needed subdirectories:

```
Storage::makeDirectory($directory);
```

Delete A Directory

Finally, the `deleteDirectory` method may be used to remove a directory and all of its files:

```
Storage::deleteDirectory($directory);
```

Custom Filesystems

Laravel's Flysystem integration provides support for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application.

In order to define a custom filesystem you will need a Flysystem adapter. Let's add a community maintained Dropbox adapter to our project:

```
composer require spatie/flysystem-dropbox
```

Next, you can register the driver within the `boot` method of one of your application's service providers. To accomplish this, you should use the `extend` method of the `Storage` facade:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorization_token']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }
}
```

The first argument of the `extend` method is the name of the driver and the second is a closure that receives the `$app` and `$config` variables. The closure must return an instance of `League\Flysystem\Filesystem`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Once you have created and registered the extension's service provider, you may use the `dropbox` driver in your `config/filesystems.php` configuration file.

Digging Deeper

Helpers

- Introduction
- Available Methods

Introduction

Laravel includes a variety of global "helper" PHP functions. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

Available Methods

Arrays & Objects

Arr::accessible Arr::add Arr::collapse Arr::crossJoin Arr::divide Arr::dot Arr::except Arr::exists Arr::first Arr::flatten Arr::forget Arr::get Arr::has Arr::hasAny Arr::isAssoc Arr::last Arr::only Arr::pluck Arr::prepend Arr::pull Arr::query Arr::random Arr::set Arr::shuffle Arr::sort Arr::sortRecursive Arr::where Arr::wrap data_fill data_get data_set head last

Paths

app_path base_path config_path database_path mix public_path resource_path storage_path

Strings

__ class_basename e preg_replace_array Str::after Str::afterLast Str::ascii Str::before Str::beforeLast Str::between Str::camel Str::contains Str::containsAll Str::endsWith Str::finish Str::is Str::isAscii Str::isUuid Str::kebab Str::length Str::limit Str::lower Str::orderedUuid Str::padBoth Str::padLeft Str::padRight Str::plural Str::random Str::replaceArray Str::replaceFirst Str::replaceLast Str::singular Str::slug Str::snake Str::start Str::startsWith Str::studly Str::substr Str::title Str::ucfirst Str::upper Str::uuid Str::words trans trans_choice

Fluent Strings

after afterLast append ascii basename before beforeLast camel contains containsAll dirname endsWith exactly explode finish is isAscii isEmpty isNotEmpty kebab length limit lower ltrim match matchAll padBoth padLeft padRight plural prepend replace replaceArray replaceFirst replaceLast replaceMatches rtrim singular slug snake split start startsWith studly substr title trim ucfirst upper when whenEmpty words

URLs

action asset route secure_asset secure_url url

Miscellaneous

abort abort_if abort_unless app auth back bcrypt blank broadcast cache class_uses_recursive collect config cookie csrf_field csrf_token dd dispatch dispatch_now dump env event filled info logger method_field now old optional policy redirect report request rescue resolve response retry session tap throw_if throw_unless today trait_uses_recursive transform validator value view with

Method Listing

Arrays & Objects

`Arr::accessible()`

The `Arr::accessible` method determines if the given value is array accessible:

```
use Illuminate\Support\Arr;
use Illuminate\Support\Collection;

$isAccessible = Arr::accessible(['a' => 1, 'b' => 2]);

// true

$isAccessible = Arr::accessible(new Collection);

// true

$isAccessible = Arr::accessible('abc');

// false

$isAccessible = Arr::accessible(new stdClass);

// false
```

`Arr::add()`

The `Arr::add` method adds a given key / value pair to an array if the given key doesn't already exist in the array or is set to `null`:

```
use Illuminate\Support\Arr;

$array = Arr::add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

`Arr::collapse()`

The `Arr::collapse` method collapses an array of arrays into a single array:

```
use Illuminate\Support\Arr;

$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`Arr::crossJoin()`

The `Arr::crossJoin` method cross joins the given arrays, returning a Cartesian product with all possible permutations:

```
use Illuminate\Support\Arr;

$matrix = Arr::crossJoin([1, 2], ['a', 'b']);

/*
    [
        [1, 'a'],
        [1, 'b'],
        [2, 'a'],
        [2, 'b'],
    ]
*/

$matrix = Arr::crossJoin([1, 2], ['a', 'b'], ['I', 'II']);
```

```
/*
    [
        [1, 'a', 'I'],
        [1, 'a', 'II'],
        [1, 'b', 'I'],
        [1, 'b', 'II'],
        [2, 'a', 'I'],
        [2, 'a', 'II'],
        [2, 'b', 'I'],
        [2, 'b', 'II'],
    ]
*/
```

`Arr::divide()`

The `Arr::divide` method returns two arrays: one containing the keys and the other containing the values of the given array:

```
use Illuminate\Support\Arr;

[$keys, $values] = Arr::divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

`Arr::dot()`

The `Arr::dot` method flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$flattened = Arr::dot($array);

// ['products.desk.price' => 100]
```

`Arr::except()`

The `Arr::except` method removes the given key / value pairs from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$filtered = Arr::except($array, ['price']);

// ['name' => 'Desk']
```

`Arr::exists()`

The `Arr::exists` method checks that the given key exists in the provided array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'John Doe', 'age' => 17];

$exists = Arr::exists($array, 'name');

// true
```

```
$exists = Arr::exists($array, 'salary');

// false
```

Arr::first()

The `Arr::first` method returns the first element of an array passing a given truth test:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300];

$first = Arr::first($array, function ($value, $key) {
    return $value >= 150;
});

// 200
```

A default value may also be passed as the third parameter to the method. This value will be returned if no value passes the truth test:

```
use Illuminate\Support\Arr;

$first = Arr::first($array, $callback, $default);
```

Arr::flatten()

The `Arr::flatten` method flattens a multi-dimensional array into a single level array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$flattened = Arr::flatten($array);

// ['Joe', 'PHP', 'Ruby']
```

Arr::forget()

The `Arr::forget` method removes a given key / value pair from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk');

// ['products' => []]
```

Arr::get()

The `Arr::get` method retrieves a value from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price');

// 100
```

The `Arr::get` method also accepts a default value, which will be returned if the specified key is not present in the array:

```
use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0);

// 0
```

Arr::has()

The `Arr::has` method checks whether a given item or items exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::has($array, 'product.name');

// true

$contains = Arr::has($array, ['product.price', 'product.discount']);

// false
```

Arr::hasAny()

The `Arr::hasAny` method checks whether any item in a given set exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::hasAny($array, 'product.name');

// true

$contains = Arr::hasAny($array, ['product.name', 'product.discount']);

// true

$contains = Arr::hasAny($array, ['category', 'product.discount']);

// false
```

Arr::isAssoc()

The `Arr::isAssoc` returns `true` if the given array is an associative array. An array is considered "associative" if it doesn't have sequential numerical keys beginning with zero:

```
use Illuminate\Support\Arr;

$isAssoc = Arr::isAssoc(['product' => ['name' => 'Desk', 'price' => 100]]);

// true

$isAssoc = Arr::isAssoc([1, 2, 3]);

// false
```

Arr::last()

The `Arr::last` method returns the last element of an array passing a given truth test:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300, 110];
```

```
$last = Arr::last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

A default value may be passed as the third argument to the method. This value will be returned if no value passes the truth test:

```
use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);
```

`Arr::only()`

The `Arr::only` method returns only the specified key / value pairs from the given array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

`Arr::pluck()`

The `Arr::pluck` method retrieves all of the values for a given key from an array:

```
use Illuminate\Support\Arr;

$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$names = Arr::pluck($array, 'developer.name');

// ['Taylor', 'Abigail']
```

You may also specify how you wish the resulting list to be keyed:

```
use Illuminate\Support\Arr;

$names = Arr::pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail']
```

`Arr::prepend()`

The `Arr::prepend` method will push an item onto the beginning of an array:

```
use Illuminate\Support\Arr;

$array = ['one', 'two', 'three', 'four'];

$array = Arr::prepend($array, 'zero');

// ['zero', 'one', 'two', 'three', 'four']
```

If needed, you may specify the key that should be used for the value:

```
use Illuminate\Support\Arr;

$array = ['price' => 100];
```

```
$array = Arr::prepend($array, 'Desk', 'name');

// ['name' => 'Desk', 'price' => 100]
```

Arr::pull()

The `Arr::pull` method returns and removes a key / value pair from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$name = Arr::pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

A default value may be passed as the third argument to the method. This value will be returned if the key doesn't exist:

```
use Illuminate\Support\Arr;

$value = Arr::pull($array, $key, $default);
```

Arr::query()

The `Arr::query` method converts the array into a query string:

```
use Illuminate\Support\Arr;

$array = [
    'name' => 'Taylor',
    'order' => [
        'column' => 'created_at',
        'direction' => 'desc'
    ]
];

Arr::query($array);

// name=Taylor&order[column]=created_at&order[direction]=desc
```

Arr::random()

The `Arr::random` method returns a random value from an array:

```
use Illuminate\Support\Arr;

$array = [1, 2, 3, 4, 5];

$random = Arr::random($array);

// 4 - (retrieved randomly)
```

You may also specify the number of items to return as an optional second argument. Note that providing this argument will return an array even if only one item is desired:

```
use Illuminate\Support\Arr;

$items = Arr::random($array, 2);

// [2, 5] - (retrieved randomly)
```

`Arr::set()`

The `Arr::set` method sets a value within a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

`Arr::shuffle()`

The `Arr::shuffle` method randomly shuffles the items in the array:

```
use Illuminate\Support\Arr;

$array = Arr::shuffle([1, 2, 3, 4, 5]);

// [3, 2, 5, 1, 4] - (generated randomly)
```

`Arr::sort()`

The `Arr::sort` method sorts an array by its values:

```
use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sort($array);

// ['Chair', 'Desk', 'Table']
```

You may also sort the array by the results of a given closure:

```
use Illuminate\Support\Arr;

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(Arr::sort($array, function ($value) {
    return $value['name'];
}));

/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
        ['name' => 'Table'],
    ]
*/
```

`Arr::sortRecursive()`

The `Arr::sortRecursive` method recursively sorts an array using the `sort` function for numerically indexed sub-arrays and the `ksort` function for associative sub-arrays:

```
use Illuminate\Support\Arr;

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
    ['one' => 1, 'two' => 2, 'three' => 3],
```

```
];

$sorted = Arr::sortRecursive($array);

/*
    [
        ['JavaScript', 'PHP', 'Ruby'],
        ['one' => 1, 'three' => 3, 'two' => 2],
        ['Li', 'Roman', 'Taylor'],
    ]
*/
```

`Arr::where()`

The `Arr::where` method filters an array using the given closure:

```
use Illuminate\Support\Arr;

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => '200', 3 => '400']
```

`Arr::wrap()`

The `Arr::wrap` method wraps the given value in an array. If the given value is already an array it be returned without modification:

```
use Illuminate\Support\Arr;

$string = 'Laravel';

$array = Arr::wrap($string);

// ['Laravel']
```

If the given value is `null`, an empty array will be returned:

```
use Illuminate\Support\Arr;

$array = Arr::wrap(null);

// []
```

`data_fill()`

The `data_fill` function sets a missing value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_fill($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 100]]]

data_fill($data, 'products.desk.discount', 10);

// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]
```

This function also accepts asterisks as wildcards and will fill the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2'],
```

```
    ],
];

data_fill($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 100],
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

data_get()

The `data_get` function retrieves a value from a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100
```

The `data_get` function also accepts a default value, which will be returned if the specified key is not found:

```
$discount = data_get($data, 'products.desk.discount', 0);

// 0
```

The function also accepts wildcards using asterisks, which may target any key of the array or object:

```
$data = [
    'product-one' => ['name' => 'Desk 1', 'price' => 100],
    'product-two' => ['name' => 'Desk 2', 'price' => 150],
];

data_get($data, '*.name');

// ['Desk 1', 'Desk 2'];
```

data_set()

The `data_set` function sets a value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

This function also accepts wildcards using asterisks and will set values on the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 150],
    ],
];

data_set($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 200],
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

By default, any existing values are overwritten. If you wish to only set a value if it doesn't exist, you may pass `false` as the fourth argument to the function:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200, $overwrite = false);

// ['products' => ['desk' => ['price' => 100]]]
```

head()

The `head` function returns the first element in the given array:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

last()

The `last` function returns the last element in the given array:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

Paths

app_path()

The `app_path` function returns the fully qualified path to your application's `app` directory. You may also use the `app_path` function to generate a fully qualified path to a file relative to the application directory:

```
$path = app_path();

$path = app_path('Http/Controllers/Controller.php');
```

base_path()

The `base_path` function returns the fully qualified path to your application's root directory. You may also use the `base_path` function to generate a fully qualified path to a given file relative to the project root directory:

```
$path = base_path();

$path = base_path('vendor/bin');
```

config_path()

The `config_path` function returns the fully qualified path to your application's `config` directory. You may also use the `config_path` function to generate a fully qualified path to a given file within the application's configuration directory:

```
$path = config_path();
```

```
$path = config_path('app.php');
```

database_path()

The `database_path` function returns the fully qualified path to your application's `database` directory. You may also use the `database_path` function to generate a fully qualified path to a given file within the database directory:

```
$path = database_path();

$path = database_path('factories/UserFactory.php');
```

mix()

The `mix` function returns the path to a versioned Mix file:

```
$path = mix('css/app.css');
```

public_path()

The `public_path` function returns the fully qualified path to your application's `public` directory. You may also use the `public_path` function to generate a fully qualified path to a given file within the public directory:

```
$path = public_path();

$path = public_path('css/app.css');
```

resource_path()

The `resource_path` function returns the fully qualified path to your application's `resources` directory. You may also use the `resource_path` function to generate a fully qualified path to a given file within the resources directory:

```
$path = resource_path();

$path = resource_path('sass/app.scss');
```

storage_path()

The `storage_path` function returns the fully qualified path to your application's `storage` directory. You may also use the `storage_path` function to generate a fully qualified path to a given file within the storage directory:

```
$path = storage_path();

$path = storage_path('app/file.txt');
```

Strings

__()

The __ function translates the given translation string or translation key using your localization files:

```
echo __('Welcome to our application');
```

```
echo __('messages.welcome');
```

If the specified translation string or key does not exist, the __ function will return the given value. So, using the example above, the __ function would return `messages.welcome` if that translation key does not exist.

class_basename()

The `class_basename` function returns the class name of the given class with the class's namespace removed:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e()

The `e` function runs PHP's `htmlspecialchars` function with the `double_encode` option set to `true` by default:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

preg_replace_array()

The `preg_replace_array` function replaces a given pattern in the string sequentially using an array:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

Str::after()

The `Str::after` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::after('This is my name', 'This is');

// ' my name'
```

Str::afterLast()

The `Str::afterLast` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::afterLast('App\Http\Controllers\Controller', '\\');

// 'Controller'
```

Str::ascii()

The `Str::ascii` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$slice = Str::ascii('û');

// 'u'
```

`Str::before()`

The `Str::before` method returns everything before the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::before('This is my name', 'my name');

// 'This is '
```

`Str::beforeLast()`

The `Str::beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::beforeLast('This is my name', 'is');

// 'This '
```

`Str::between()`

The `Str::between` method returns the portion of a string between two values:

```
use Illuminate\Support\Str;

$slice = Str::between('This is my name', 'This', 'name');

// ' is my '
```

`Str::camel()`

The `Str::camel` method converts the given string to `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::camel('foo_bar');

// fooBar
```

`Str::contains()`

The `Str::contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'my');

// true
```

You may also pass an array of values to determine if the given string contains any of the values in the array:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', ['my', 'foo']);

// true
```

`Str::containsAll()`

The `Str::containsAll` method determines if the given string contains all of the values in a given array:

```
use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['my', 'name']);

// true
```

`Str::endsWith()`

The `Str::endsWith` method determines if the given string ends with the given value:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', 'name');

// true
```

You may also pass an array of values to determine if the given string ends with any of the values in the array:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', ['name', 'foo']);

// true

$result = Str::endsWith('This is my name', ['this', 'foo']);

// false
```

`Str::finish()`

The `Str::finish` method adds a single instance of the given value to a string if it does not already end with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::finish('this/string', '/');

// this/string/

$adjusted = Str::finish('this/string/', '/');

// this/string/
```

`Str::is()`

The `Str::is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values:

```
use Illuminate\Support\Str;

$matches = Str::is('foo*', 'foobar');
```

```
// true
$matches = Str::is('baz*', 'foobar');
// false
```

Str::isAscii()

The `Str::isAscii` method determines if a given string is 7 bit ASCII:

```
use Illuminate\Support\Str;
$isAscii = Str::isAscii('Taylor');
// true
$isAscii = Str::isAscii('ü');
// false
```

Str::isUuid()

The `Str::isUuid` method determines if the given string is a valid UUID:

```
use Illuminate\Support\Str;
$isUuid = Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');
// true
$isUuid = Str::isUuid('laravel');
// false
```

Str::kebab()

The `Str::kebab` method converts the given string to `kebab-case`:

```
use Illuminate\Support\Str;
$converted = Str::kebab('fooBar');
// foo-bar
```

Str::length()

The `Str::length` method returns the length of the given string:

```
use Illuminate\Support\Str;
$length = Str::length('Laravel');
// 7
```

Str::limit()

The `Str::limit` method truncates the given string to the specified length:

```
use Illuminate\Support\Str;
```

```
$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20);

// The quick brown fox...
```

You may pass a third argument to the method to change the string that will be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20, ' (...)');

// The quick brown fox (...)
```

`Str::lower()`

The `Str::lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$converted = Str::lower('LARAVEL');

// laravel
```

`Str::orderedUuid()`

The `Str::orderedUuid` method generates a "timestamp first" UUID that may be efficiently stored in an indexed database column. Each UUID that is generated using this method will be sorted after UUIDs previously generated using the method:

```
use Illuminate\Support\Str;

return (string) Str::orderedUuid();
```

`Str::padBoth()`

The `Str::padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padBoth('James', 10, '_');

// '__James___'

$padded = Str::padBoth('James', 10);

// '  James   '
```

`Str::padLeft()`

The `Str::padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padLeft('James', 10, '-=');

// '-=-=-James'

$padded = Str::padLeft('James', 10);

// '     James'
```

`Str::padRight()`

The `Str::padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padRight('James', 10, '-');

// 'James-----'

$padded = Str::padRight('James', 10);

// 'James     '
```

`Str::plural()`

The `Str::plural` method converts a singular word string to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::plural('car');

// cars

$plural = Str::plural('child');

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::plural('child', 2);

// children

$plural = Str::plural('child', 1);

// child
```

`Str::random()`

The `Str::random` method generates a random string of the specified length. This function uses PHP's `random_bytes` function:

```
use Illuminate\Support\Str;

$random = Str::random(40);
```

`Str::replaceArray()`

The `Str::replaceArray` method replaces a given value in the string sequentially using an array:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

`Str::replaceFirst()`

The `Str::replaceFirst` method replaces the first occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps over the lazy dog');

// a quick brown fox jumps over the lazy dog
```

`Str::replaceLast()`

The `Str::replaceLast` method replaces the last occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over the lazy dog');

// the quick brown fox jumps over a lazy dog
```

`Str::singular()`

The `Str::singular` method converts a string to its singular form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$singular = Str::singular('cars');

// car

$singular = Str::singular('children');

// child
```

`Str::slug()`

The `Str::slug` method generates a URL friendly "slug" from the given string:

```
use Illuminate\Support\Str;

$slug = Str::slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

`Str::snake()`

The `Str::snake` method converts the given string to `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::snake('fooBar');

// foo_bar
```

`Str::start()`

The `Str::start` method adds a single instance of the given value to a string if it does not already start with that value:

```
use Illuminate\Support\Str;
$adjusted = Str::start('this/string', '/');
// /this/string
$adjusted = Str::start('/this/string', '/');
// /this/string
```

`Str::startsWith()`

The `Str::startsWith` method determines if the given string begins with the given value:

```
use Illuminate\Support\Str;
$result = Str::startsWith('This is my name', 'This');
// true
```

`Str::studly()`

The `Str::studly` method converts the given string to `StudlyCase`:

```
use Illuminate\Support\Str;
$converted = Str::studly('foo_bar');
// FooBar
```

`Str::substr()`

The `Str::substr` method returns the portion of string specified by the start and length parameters:

```
use Illuminate\Support\Str;
$converted = Str::substr('The Laravel Framework', 4, 7);
// Laravel
```

`Str::title()`

The `Str::title` method converts the given string to `Title Case`:

```
use Illuminate\Support\Str;
$converted = Str::title('a nice title uses the correct case');
// A Nice Title Uses The Correct Case
```

`Str::ucfirst()`

The `Str::ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;
$string = Str::ucfirst('foo bar');
// Foo bar
```

`Str::upper()`

The `Str::upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$string = Str::upper('laravel');

// LARAVEL
```

`Str::uuid()`

The `Str::uuid` method generates a UUID (version 4):

```
use Illuminate\Support\Str;

return (string) Str::uuid();
```

`Str::words()`

The `Str::words` method limits the number of words in a string. An additional string may be passed to this method via its third argument to specify which string should be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

return Str::words('Perfectly balanced, as all things should be.', 3, ' >>>');

// Perfectly balanced, as >>>
```

`trans()`

The `trans` function translates the given translation key using your localization files:

```
echo trans('messages.welcome');
```

If the specified translation key does not exist, the `trans` function will return the given key. So, using the example above, the `trans` function would return `messages.welcome` if the translation key does not exist.

`trans_choice()`

The `trans_choice` function translates the given translation key with inflection:

```
echo trans_choice('messages.notifications', $unreadCount);
```

If the specified translation key does not exist, the `trans_choice` function will return the given key. So, using the example above, the `trans_choice` function would return `messages.notifications` if the translation key does not exist.

Fluent Strings

Fluent strings provide a more fluent, object-oriented interface for working with string values, allowing you to chain multiple string operations together using a more readable syntax compared to traditional string operations.

`after`

The `after` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->after('This is');

// ' my name'
```

afterLast

The `afterLast` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('App\Http\Controllers\Controller')->afterLast('\\');

// 'Controller'
```

append

The `append` method appends the given values to the string:

```
use Illuminate\Support\Str;

$string = Str::of('Taylor')->append(' Otwell');

// 'Taylor Otwell'
```

ascii

The `ascii` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$string = Str::of('ü')->ascii();

// 'u'
```

basename

The `basename` method will return the trailing name component of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->basename();

// 'baz'
```

If needed, you may provide an "extension" that will be removed from the trailing component:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz.jpg')->basename('.jpg');

// 'baz'
```

before

The `before` method returns everything before the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->before('my name');

// 'This is '
```

beforeLast

The `beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->beforeLast('is');

// 'This '
```

camel

The `camel` method converts the given string to `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::of('foo_bar')->camel();

// fooBar
```

contains

The `contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains('my');

// true
```

You may also pass an array of values to determine if the given string contains any of the values in the array:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains(['my', 'foo']);

// true
```

containsAll

The `containsAll` method determines if the given string contains all of the values in the given array:

```
use Illuminate\Support\Str;

$containsAll = Str::of('This is my name')->containsAll(['my', 'name']);

// true
```

dirname

The `dirname` method returns the parent directory portion of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname();

// '/foo/bar'
```

If necessary, you may specify how many directory levels you wish to trim from the string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname(2);

// '/foo'
```

endsWith

The `endsWith` method determines if the given string ends with the given value:

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith('name');

// true
```

You may also pass an array of values to determine if the given string ends with any of the values in the array:

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith(['name', 'foo']);

// true

$result = Str::of('This is my name')->endsWith(['this', 'foo']);

// false
```

exactly

The `exactly` method determines if the given string is an exact match with another string:

```
use Illuminate\Support\Str;

$result = Str::of('Laravel')->exactly('Laravel');

// true
```

explode

The `explode` method splits the string by the given delimiter and returns a collection containing each section of the split string:

```
use Illuminate\Support\Str;

$collection = Str::of('foo bar baz')->explode(' ');

// collect(['foo', 'bar', 'baz'])
```

finish

The `finish` method adds a single instance of the given value to a string if it does not already end with that value:

```
use Illuminate\Support\Str;
```

```
$adjusted = Str::of('this/string')->finish('/');
// this/string/
$adjusted = Str::of('this/string/')->finish('/');
// this/string/
```

is

The `is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values

```
use Illuminate\Support\Str;
$matches = Str::of('foobar')->is('foo*');
// true
$matches = Str::of('foobar')->is('baz*');
// false
```

isAscii

The `isAscii` method determines if a given string is an ASCII string:

```
use Illuminate\Support\Str;
$result = Str::of('Taylor')->isAscii();
// true
$result = Str::of('ü')->isAscii();
// false
```

isEmpty

The `isEmpty` method determines if the given string is empty:

```
use Illuminate\Support\Str;
$result = Str::of('  ')->trim()->isEmpty();
// true
$result = Str::of('Laravel')->trim()->isEmpty();
// false
```

isNotEmpty

The `isNotEmpty` method determines if the given string is not empty:

```
use Illuminate\Support\Str;
$result = Str::of('  ')->trim()->isNotEmpty();
// false
$result = Str::of('Laravel')->trim()->isNotEmpty();
// true
```

kebab

The `kebab` method converts the given string to `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->kebab();

// foo-bar
```

length

The `length` method returns the length of the given string:

```
use Illuminate\Support\Str;

$length = Str::of('Laravel')->length();

// 7
```

limit

The `limit` method truncates the given string to the specified length:

```
use Illuminate\Support\Str;

$truncated = Str::of('The quick brown fox jumps over the lazy dog')->limit(20);

// The quick brown fox...
```

You may also pass a second argument to change the string that will be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

$truncated = Str::of('The quick brown fox jumps over the lazy dog')->limit(20, ' (...)');

// The quick brown fox (...)
```

lower

The `lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$result = Str::of('LARAVEL')->lower();

// 'laravel'
```

ltrim

The `ltrim` method trims the left side of the string:

```
use Illuminate\Support\Str;

$string = Str::of('  Laravel  ')->ltrim();

// 'Laravel  '

$string = Str::of('/Laravel/')->ltrim('/');

// 'Laravel/'
```

match

The `match` method will return the portion of a string that matches a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('foo bar')->match('/bar/');

// 'bar'

$result = Str::of('foo bar')->match('/foo (.*)/');

// 'bar'
```

matchAll

The `matchAll` method will return a collection containing the portions of a string that match a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('bar foo bar')->matchAll('/bar/');

// collect(['bar', 'bar'])
```

If you specify a matching group within the expression, Laravel will return a collection of that group's matches:

```
use Illuminate\Support\Str;

$result = Str::of('bar fun bar fly')->matchAll('/f(\w*)/');

// collect(['un', 'ly']);
```

If no matches are found, an empty collection will be returned.

padBoth

The `padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padBoth(10, '_');

// '__James___'

$padded = Str::of('James')->padBoth(10);

// '  James   '
```

padLeft

The `padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padLeft(10, '-=');

// '-=-=-James'

$padded = Str::of('James')->padLeft(10);

// '     James'
```

padRight

The `padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padRight(10, '-');

// 'James-----'

$padded = Str::of('James')->padRight(10);

// 'James     '
```

plural

The `plural` method converts a singular word string to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::of('car')->plural();

// cars

$plural = Str::of('child')->plural();

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::of('child')->plural(2);

// children

$plural = Str::of('child')->plural(1);

// child
```

prepend

The `prepend` method prepends the given values onto the string:

```
use Illuminate\Support\Str;

$string = Str::of('Framework')->prepend('Laravel ');

// Laravel Framework
```

replace

The `replace` method replaces a given string within the string:

```
use Illuminate\Support\Str;

$replaced = Str::of('Laravel 6.x')->replace('6.x', '7.x');

// Laravel 7.x
```

replaceArray

The `replaceArray` method replaces a given value in the string sequentially using an array:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::of($string)->replaceArray('?', ['8:30', '9:00']);

// The event will take place between 8:30 and 9:00
```

replaceFirst

The `replaceFirst` method replaces the first occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceFirst('the', 'a');

// a quick brown fox jumps over the lazy dog
```

replaceLast

The `replaceLast` method replaces the last occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceLast('the', 'a');

// the quick brown fox jumps over a lazy dog
```

replaceMatches

The `replaceMatches` method replaces all portions of a string matching a pattern with the given replacement string:

```
use Illuminate\Support\Str;

$replaced = Str::of('(+1) 501-555-1000')->replaceMatches('/[^A-Za-z0-9]++/', '')

// '15015551000'
```

The `replaceMatches` method also accepts a closure that will be invoked with each portion of the string matching the given pattern, allowing you to perform the replacement logic within the closure and return the replaced value:

```
use Illuminate\Support\Str;

$replaced = Str::of('123')->replaceMatches('/\d/', function ($match) {
    return '['.$match[0].']';
});

// '[1][2][3]'
```

rtrim

The `rtrim` method trims the right side of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('  Laravel  ')->rtrim();
```

```
// '  Laravel'

$string = Str::of('/Laravel/')->rtrim('/');

// '/Laravel'
```

singular

The `singular` method converts a string to its singular form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$singular = Str::of('cars')->singular();

// car

$singular = Str::of('children')->singular();

// child
```

slug

The `slug` method generates a URL friendly "slug" from the given string:

```
use Illuminate\Support\Str;

$slug = Str::of('Laravel Framework')->slug('-');

// laravel-framework
```

snake

The `snake` method converts the given string to `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->snake();

// foo_bar
```

split

The `split` method splits a string into a collection using a regular expression:

```
use Illuminate\Support\Str;

$segments = Str::of('one, two, three')->split('/[\s,]+/');

// collect(["one", "two", "three"])
```

start

The `start` method adds a single instance of the given value to a string if it does not already start with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->start('/');
```

```
// /this/string
$adjusted = Str::of('/this/string')->start('/');
// /this/string
```

startsWith

The `startsWith` method determines if the given string begins with the given value:

```
use Illuminate\Support\Str;
$result = Str::of('This is my name')->startsWith('This');
// true
```

studly

The `studly` method converts the given string to `StudlyCase`:

```
use Illuminate\Support\Str;
$converted = Str::of('foo_bar')->studly();
// FooBar
```

substr

The `substr` method returns the portion of the string specified by the given start and length parameters:

```
use Illuminate\Support\Str;
$string = Str::of('Laravel Framework')->substr(8);
// Framework
$string = Str::of('Laravel Framework')->substr(8, 5);
// Frame
```

title

The `title` method converts the given string to `Title Case`:

```
use Illuminate\Support\Str;
$converted = Str::of('a nice title uses the correct case')->title();
// A Nice Title Uses The Correct Case
```

trim

The `trim` method trims the given string:

```
use Illuminate\Support\Str;
$string = Str::of('  Laravel  ')->trim();
// 'Laravel'
$string = Str::of('/Laravel/')->trim('/');
```

```
// 'Laravel'
```

ucfirst

The `ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;

$string = Str::of('foo bar')->ucfirst();

// Foo bar
```

upper

The `upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$adjusted = Str::of('laravel')->upper();

// LARAVEL
```

when

The `when` method invokes the given closure if a given condition is `true`. The closure will receive the fluent string instance:

```
use Illuminate\Support\Str;

$string = Str::of('Taylor')
                ->when(true, function ($string) {
                    return $string->append(' Otwell');
                });

// 'Taylor Otwell'
```

If necessary, you may pass another closure as the third parameter to the `when` method. This closure will execute if the condition parameter evaluates to `false`.

whenEmpty

The `whenEmpty` method invokes the given closure if the string is empty. If the closure returns a value, that value will also be returned by the `whenEmpty` method. If the closure does not return a value, the fluent string instance will be returned:

```
use Illuminate\Support\Str;

$string = Str::of('  ')->whenEmpty(function ($string) {
    return $string->trim()->prepend('Laravel');
});

// 'Laravel'
```

words

The `words` method limits the number of words in a string. If necessary, you may specify an additional string that will be appended to the truncated string:

```
use Illuminate\Support\Str;

$string = Str::of('Perfectly balanced, as all things should be.')->words(3, ' >>>');

// Perfectly balanced, as >>>
```

URLs

action()

The `action` function generates a URL for the given controller action:

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

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

If the method accepts route parameters, you may pass them as the second argument to the method:

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

asset()

The `asset` function generates a URL for an asset using the current scheme of the request (HTTP or HTTPS):

```
$url = asset('img/photo.jpg');
```

You can configure the asset URL host by setting the `ASSET_URL` variable in your `.env` file. This can be useful if you host your assets on an external service like Amazon S3 or another CDN:

```
// ASSET_URL=http://example.com/assets

$url = asset('img/photo.jpg'); // http://example.com/assets/img/photo.jpg
```

route()

The `route` function generates a URL for a given named route:

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

If the route accepts parameters, you may pass them as the second argument to the function:

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

By default, the `route` function generates an absolute URL. If you wish to generate a relative URL, you may pass `false` as the third argument to the function:

```
$url = route('route.name', ['id' => 1], false);
```

secure_asset()

The `secure_asset` function generates a URL for an asset using HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

secure_url()

The `secure_url` function generates a fully qualified HTTPS URL to the given path. Additional URL segments may be passed in the function's second argument:

```
$url = secure_url('user/profile');

$url = secure_url('user/profile', [1]);
```

`url()`

The `url` function generates a fully qualified URL to the given path:

```
$url = url('user/profile');

$url = url('user/profile', [1]);
```

If no path is provided, an `Illuminate\Routing\UrlGenerator` instance is returned:

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

$full = url()->full();

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

Miscellaneous

`abort()`

The `abort` function throws an HTTP exception which will be rendered by the exception handler:

```
abort(403);
```

You may also provide the exception's message and custom HTTP response headers that should be sent to the browser:

```
abort(403, 'Unauthorized.', $headers);
```

`abort_if()`

The `abort_if` function throws an HTTP exception if a given boolean expression evaluates to `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument to the function.

`abort_unless()`

The `abort_unless` function throws an HTTP exception if a given boolean expression evaluates to `false`:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument to the function.

`app()`

The `app` function returns the service container instance:

```
$container = app();
```

You may pass a class or interface name to resolve it from the container:

```
$api = app('HelpSpot\API');
```

auth()

The `auth` function returns an authenticator instance. You may use it as an alternative to the `Auth` facade:

```
$user = auth()->user();
```

If needed, you may specify which guard instance you would like to access:

```
$user = auth('admin')->user();
```

back()

The `back` function generates a redirect HTTP response to the user's previous location:

```
return back($status = 302, $headers = [], $fallback = '/');

return back();
```

bcrypt()

The `bcrypt` function hashes the given value using Bcrypt. You may use this function as an alternative to the `Hash` facade:

```
$password = bcrypt('my-secret-password');
```

blank()

The `blank` function determines whether the given value is "blank":

```
blank('');
blank('   ');
blank(null);
blank(collect());

// true

blank(0);
blank(true);
blank(false);

// false
```

For the inverse of `blank`, see the `filled` method.

broadcast()

The `broadcast` function broadcasts the given event to its listeners:

```
broadcast(new UserRegistered($user));
```

```
broadcast(new UserRegistered($user))->toOthers();
```

cache()

The `cache` function may be used to get values from the cache. If the given key does not exist in the cache, an optional default value will be returned:

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

$value = cache('key', 'default');
```

You may add items to the cache by passing an array of key / value pairs to the function. You should also pass the number of seconds or duration the cached value should be considered valid:

```
cache(['key' => 'value'], 300);

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

class_uses_recursive()

The `class_uses_recursive` function returns all traits used by a class, including traits used by all of its parent classes:

```
$traits = class_uses_recursive(App\Models\User::class);
```

collect()

The `collect` function creates a collection instance from the given value:

```
$collection = collect(['taylor', 'abigail']);
```

config()

The `config` function gets the value of a configuration variable. The configuration values may be accessed using "dot" syntax, which includes the name of the file and the option you wish to access. A default value may be specified and is returned if the configuration option does not exist:

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

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

You may set configuration variables at runtime by passing an array of key / value pairs. However, note that this function only affects the configuration value for the current request and does not update your actual configuration values:

```
config(['app.debug' => true]);
```

cookie()

The `cookie` function creates a new cookie instance:

```
$cookie = cookie('name', 'value', $minutes);
```

csrf_field()

The `csrf_field` function generates an HTML `hidden` input field containing the value of the CSRF token. For example, using Blade syntax:

```
{{ csrf_field() }}
```

csrf_token()

The `csrf_token` function retrieves the value of the current CSRF token:

```
$token = csrf_token();
```

dd()

The `dd` function dumps the given variables and ends execution of the script:

```
dd($value);

dd($value1, $value2, $value3, ...);
```

If you do not want to halt the execution of your script, use the `dump` function instead.

dispatch()

The `dispatch` function pushes the given job onto the Laravel job queue:

```
dispatch(new App\Jobs\SendEmails);
```

dispatch_now()

The `dispatch_now` function runs the given job immediately and returns the value from its `handle` method:

```
$result = dispatch_now(new App\Jobs\SendEmails);
```

dump()

The `dump` function dumps the given variables:

```
dump($value);

dump($value1, $value2, $value3, ...);
```

If you want to stop executing the script after dumping the variables, use the `dd` function instead.

env()

The `env` function retrieves the value of an environment variable or returns a default value:

```
$env = env('APP_ENV');

$env = env('APP_ENV', 'production');
```

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

event()

The `event` function dispatches the given event to its listeners:

```
event(new UserRegistered($user));
```

filled()

The `filled` function determines whether the given value is not "blank":

```
filled(0);
filled(true);
filled(false);

// true

filled('');
filled('   ');
filled(null);
filled(collect());

// false
```

For the inverse of `filled`, see the `blank` method.

info()

The `info` function will write information to your application's log:

```
info('Some helpful information!');
```

An array of contextual data may also be passed to the function:

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

logger()

The `logger` function can be used to write a `debug` level message to the log:

```
logger('Debug message');
```

An array of contextual data may also be passed to the function:

```
logger('User has logged in.', ['id' => $user->id]);
```

A logger instance will be returned if no value is passed to the function:

```
logger()->error('You are not allowed here.');
```

method_field()

The `method_field` function generates an HTML `hidden` input field containing the spoofed value of the form's HTTP verb. For example, using Blade syntax:

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

now()

The `now` function creates a new `Illuminate\Support\Carbon` instance for the current time:

```
$now = now();
```

old()

The `old` function retrieves an old input value flashed into the session:

```
$value = old('value');

$value = old('value', 'default');
```

optional()

The `optional` function accepts any argument and allows you to access properties or call methods on that object. If the given object is `null`, properties and methods will return `null` instead of causing an error:

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

The `optional` function also accepts a closure as its second argument. The closure will be invoked if the value provided as the first argument is not null:

```
return optional(User::find($id), function ($user) {
    return $user->name;
});
```

policy()

The `policy` method retrieves a policy instance for a given class:

```
$policy = policy(App\Models\User::class);
```

redirect()

The `redirect` function returns a redirect HTTP response, or returns the redirector instance if called with no arguments:

```
return redirect($to = null, $status = 302, $headers = [], $https = null);

return redirect('/home');

return redirect()->route('route.name');
```

report()

The `report` function will report an exception using your exception handler:

```
report($e);
```

request()

The `request` function returns the current request instance or obtains an input field's value from the current request:

```
$request = request();

$value = request('key', $default);
```

rescue()

The `rescue` function executes the given closure and catches any exceptions that occur during its execution. All exceptions that are caught will be sent to your exception handler; however, the request will continue processing:

```
return rescue(function () {
    return $this->method();
});
```

You may also pass a second argument to the `rescue` function. This argument will be the "default" value that should be returned if an exception occurs while executing the closure:

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

resolve()

The `resolve` function resolves a given class or interface name to an instance using the service container:

```
$api = resolve('HelpSpot\API');
```

response()

The `response` function creates a response instance or obtains an instance of the response factory:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

retry()

The `retry` function attempts to execute the given callback until the given maximum attempt threshold is met. If the callback does not throw an exception, its return value will be returned. If the callback throws an exception, it will automatically be retried. If the maximum attempt count is exceeded, the exception will be thrown:

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

session()

The `session` function may be used to get or set session values:

```
$value = session('key');
```

You may set values by passing an array of key / value pairs to the function:

```
session(['chairs' => 7, 'instruments' => 3]);
```

The session store will be returned if no value is passed to the function:

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

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

tap()

The `tap` function accepts two arguments: an arbitrary `$value` and a closure. The `$value` will be passed to the closure and then be returned by the `tap` function. The return value of the closure is irrelevant:

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';

    $user->save();
});
```

If no closure is passed to the `tap` function, you may call any method on the given `$value`. The return value of the method you call will always be `$value`, regardless of what the method actually returns in its definition. For example, the Eloquent `update` method typically returns an integer. However, we can force the method to return the model itself by chaining the `update` method call through the `tap` function:

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

To add a `tap` method to a class, you may add the `Illuminate\Support\Traits\Tappable` trait to the class. The `tap` method of this trait accepts a Closure as its only argument. The object instance itself will be passed to the Closure and then be returned by the `tap` method:

```
return $user->tap(function ($user) {
    //
});
```

throw_if()

The `throw_if` function throws the given exception if a given boolean expression evaluates to `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page.'
);
```

throw_unless()

The `throw_unless` function throws the given exception if a given boolean expression evaluates to `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page.'
);
```

today()

The `today` function creates a new `Illuminate\Support\Carbon` instance for the current date:

```
$today = today();
```

trait_uses_recursive()

The `trait_uses_recursive` function returns all traits used by a trait:

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

transform()

The `transform` function executes a closure on a given value if the value is not blank and then returns the return value of the closure:

```
$callback = function ($value) {
    return $value * 2;
};

$result = transform(5, $callback);

// 10
```

A default value or closure may be passed as the third argument to the function. This value will be returned if the given value is blank:

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

validator()

The `validator` function creates a new validator instance with the given arguments. You may use it as an alternative to the `Validator` facade:

```
$validator = validator($data, $rules, $messages);
```

value()

The `value` function returns the value it is given. However, if you pass a closure to the function, the closure will be executed and its returned value will be returned:

```
$result = value(true);

// true

$result = value(function () {
    return false;
});

// false
```

view()

The `view` function retrieves a view instance:

```
return view('auth.login');
```

with()

The `with` function returns the value it is given. If a closure is passed as the second argument to the function, the closure will be executed and its returned value will be returned:

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

Digging Deeper

HTTP Client

- Introduction
- Making Requests
  - Request Data
  - Headers
  - Authentication
  - Timeout
  - Retries
  - Error Handling
  - Guzzle Options
- Testing
  - Faking Responses
  - Inspecting Requests

Introduction

Laravel provides an expressive, minimal API around the Guzzle HTTP client, allowing you to quickly make outgoing HTTP requests to communicate with other web applications. Laravel's wrapper around Guzzle is focused on its most common use cases and a wonderful developer experience.

Before getting started, you should ensure that you have installed the Guzzle package as a dependency of your application. By default, Laravel automatically includes this dependency. However, if you have previously removed the package, you may install it again via Composer:

```
composer require guzzlehttp/guzzle
```

Making Requests

To make requests, you may use the `get`, `post`, `put`, `patch`, and `delete` methods provided by the `Http` facade. First, let's examine how to make a basic `GET` request to another URL:

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

The `get` method returns an instance of `Illuminate\Http\Client\Response`, which provides a variety of methods that may be used to inspect the response:

```
$response->body() : string;
$response->json() : array|mixed;
$response->status() : int;
$response->ok() : bool;
$response->successful() : bool;
$response->failed() : bool;
$response->serverError() : bool;
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

The `Illuminate\Http\Client\Response` object also implements the PHP `ArrayAccess` interface, allowing you to access JSON response data directly on the response:

```
return Http::get('http://example.com/users/1')['name'];
```

Request Data

Of course, it is common when making `POST`, `PUT`, and `PATCH` requests to send additional data with your request, so these methods accept an array of data as their second argument. By default, data will be sent using the `application/json` content type:

```
use Illuminate\Support\Facades\Http;

$response = Http::post('http://example.com/users', [
    'name' => 'Steve',
    'role' => 'Network Administrator',
]);
```

GET Request Query Parameters

When making `GET` requests, you may either append a query string to the URL directly or pass an array of key / value pairs as the second argument to the `get` method:

```
$response = Http::get('http://example.com/users', [
    'name' => 'Taylor',
    'page' => 1,
]);
```

Sending Form URL Encoded Requests

If you would like to send data using the `application/x-www-form-urlencoded` content type, you should call the `asForm` method before making your request:

```
$response = Http::asForm()->post('http://example.com/users', [
    'name' => 'Sara',
    'role' => 'Privacy Consultant',
]);
```

Sending A Raw Request Body

You may use the `withBody` method if you would like to provide a raw request body when making a request. The content type may be provided via the method's second argument:

```
$response = Http::withBody(
    base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

Multi-Part Requests

If you would like to send files as multi-part requests, you should call the `attach` method before making your request. This method accepts the name of the file and its contents. If needed, you may provide a third argument which will be considered the file's filename:

```
$response = Http::attach(
    'attachment', file_get_contents('photo.jpg'), 'photo.jpg'
)->post('http://example.com/attachments');
```

Instead of passing the raw contents of a file, you may pass a stream resource:

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
    'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

Headers

Headers may be added to requests using the `withHeaders` method. This `withHeaders` method accepts an array of key / value pairs:

```
$response = Http::withHeaders([
    'X-First' => 'foo',
    'X-Second' => 'bar'
])->post('http://example.com/users', [
    'name' => 'Taylor',
]);
```

Authentication

You may specify basic and digest authentication credentials using the `withBasicAuth` and `withDigestAuth` methods, respectively:

```
// Basic authentication...
$response = Http::withBasicAuth('taylor@laravel.com', 'secret')->post(...);

// Digest authentication...
$response = Http::withDigestAuth('taylor@laravel.com', 'secret')->post(...);
```

Bearer Tokens

If you would like to quickly add a bearer token to the request's `Authorization` header, you may use the `withToken` method:

```
$response = Http::withToken('token')->post(...);
```

Timeout

The `timeout` method may be used to specify the maximum number of seconds to wait for a response:

```
$response = Http::timeout(3)->get(...);
```

If the given timeout is exceeded, an instance of `Illuminate\Http\Client\ConnectionException` will be thrown.

Retries

If you would like HTTP client to automatically retry the request if a client or server error occurs, you may use the `retry` method. The `retry` method accepts two arguments: the maximum number of times the request should be attempted and the number of milliseconds that Laravel should wait in between attempts:

```
$response = Http::retry(3, 100)->post(...);
```

If all of the requests fail, an instance of `Illuminate\Http\Client\RequestException` will be thrown.

Error Handling

Unlike Guzzle's default behavior, Laravel's HTTP client wrapper does not throw exceptions on client or server errors (`400` and `500` level responses from servers). You may determine if one of these errors was returned using the `successful`, `clientError`, or `serverError` methods:

```
// Determine if the status code is >= 200 and < 300...
$response->successful();

// Determine if the status code is >= 400...
$response->failed();

// Determine if the response has a 400 level status code...
$response->clientError();

// Determine if the response has a 500 level status code...
$response->serverError();
```

Throwing Exceptions

If you have a response instance and would like to throw an instance of `Illuminate\Http\Client\RequestException` if the response status code indicates a client or server error, you may use the `throw` method:

```
$response = Http::post(...);

// Throw an exception if a client or server error occurred...
$response->throw();

return $response['user']['id'];
```

The `Illuminate\Http\Client\RequestException` instance has a public `$response` property which will allow you to

inspect the returned response.

The `throw` method returns the response instance if no error occurred, allowing you to chain other operations onto the `throw` method:

```
return Http::post(...)->throw()->json();
```

If you would like to perform some additional logic before the exception is thrown, you may pass a closure to the `throw` method. The exception will be thrown automatically after the closure is invoked, so you do not need to re-throw the exception from within the closure:

```
return Http::post(...)->throw(function ($response, $e) {
    //
})->json();
```

Guzzle Options

You may specify additional Guzzle request options using the `withOptions` method. The `withOptions` method accepts an array of key / value pairs:

```
$response = Http::withOptions([
    'debug' => true,
])->get('http://example.com/users');
```

Testing

Many Laravel services provide functionality to help you easily and expressively write tests, and Laravel's HTTP wrapper is no exception. The `Http` facade's `fake` method allows you to instruct the HTTP client to return stubbed / dummy responses when requests are made.

Faking Responses

For example, to instruct the HTTP client to return empty, `200` status code responses for every request, you may call the `fake` method with no arguments:

```
use Illuminate\Support\Facades\Http;

Http::fake();

$response = Http::post(...);
```

> NOTE When faking requests, HTTP client middleware are not executed. You should define expectations for faked responses as if these middleware have run correctly.

Faking Specific URLs

Alternatively, you may pass an array to the `fake` method. The array's keys should represent URL patterns that you wish to fake and their associated responses. The `*` character may be used as a wildcard character. Any requests made to URLs that have not been faked will actually be executed. You may use the `Http` facade's `response` method to construct stub / fake responses for these endpoints:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

    // Stub a string response for Google endpoints...
    'google.com/*' => Http::response('Hello World', 200, $headers),
]);
```

If you would like to specify a fallback URL pattern that will stub all unmatched URLs, you may use a single `*` character:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, ['Headers']),
```

```
    // Stub a string response for all other endpoints...
    '*' => Http::response('Hello World', 200, ['Headers']),
]);
```

Faking Response Sequences

Sometimes you may need to specify that a single URL should return a series of fake responses in a specific order. You may accomplish this using the `Http::sequence` method to build the responses:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->pushStatus(404),
]);
```

When all of the responses in a response sequence have been consumed, any further requests will cause the response sequence to throw an exception. If you would like to specify a default response that should be returned when a sequence is empty, you may use the `whenEmpty` method:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->whenEmpty(Http::response()),
]);
```

If you would like to fake a sequence of responses but do not need to specify a specific URL pattern that should be faked, you may use the `Http::fakeSequence` method:

```
Http::fakeSequence()
        ->push('Hello World', 200)
        ->whenEmpty(Http::response());
```

Fake Callback

If you require more complicated logic to determine what responses to return for certain endpoints, you may pass a closure to the `fake` method. This closure will receive an instance of `Illuminate\Http\Client\Request` and should return a response instance. Within your closure, you may perform whatever logic is necessary to determine what type of response to return:

```
Http::fake(function ($request) {
    return Http::response('Hello World', 200);
});
```

Inspecting Requests

When faking responses, you may occasionally wish to inspect the requests the client receives in order to make sure your application is sending the correct data or headers. You may accomplish this by calling the `Http::assertSent` method after calling `Http::fake`.

The `assertSent` method accepts a closure which will receive an `Illuminate\Http\Client\Request` instance and should return a boolean value indicating if the request matches your expectations. In order for the test to pass, at least one request must have been issued matching the given expectations:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
    'X-First' => 'foo',
])->post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
```

```
    return $request->hasHeader('X-First', 'foo') &&
           $request->url() == 'http://example.com/users' &&
           $request['name'] == 'Taylor' &&
           $request['role'] == 'Developer';
});
```

If needed, you may assert that a specific request was not sent using the `assertNotSent` method:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

Or, you may use the `assertNothingSent` method to assert that no requests were sent during the test:

```
Http::fake();

Http::assertNothingSent();
```

Digging Deeper

Localization

- Introduction
  - Configuring The Locale
- Defining Translation Strings
  - Using Short Keys
  - Using Translation Strings As Keys
- Retrieving Translation Strings
  - Replacing Parameters In Translation Strings
  - Pluralization
- Overriding Package Language Files

Introduction

Laravel's localization features provide a convenient way to retrieve strings in various languages, allowing you to easily support multiple languages within your application.

Laravel provides two ways to manage translation strings. First, language strings may be stored in files within the `resources/lang` directory. Within this directory there may be subdirectories for each language supported by the application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

Or, translation strings may be defined within JSON files that are placed within the `resources/lang` directory. When taking this approach, each language supported by your application would have a corresponding JSON file within this directory. This approach is recommended for application's that have a large number of translatable strings:

```
/resources
    /lang
        en.json
        es.json
```

We'll discuss each approach to managing translation strings within this documentation.

Configuring The Locale

The default language for your application is stored in the `config/app.php` configuration file's `locale` configuration option. You are free to modify this value to suit the needs of your application.

You may modify the default language for a single HTTP request at runtime using the `setLocale` method provided by the `App` facade:

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

Route::get('/greeting/{locale}', function ($locale) {
    if (! in_array($locale, ['en', 'es', 'fr'])) {
        abort(400);
    }

    App::setLocale($locale);

    //
});
```

You may configure a "fallback language", which will be used when the active language does not contain a given translation string. Like the default language, the fallback language is also configured in the

`config/app.php` configuration file:

```
'fallback_locale' => 'en',
```

Determining The Current Locale

You may use the `currentLocale` and `isLocale` methods on the `App` facade to determine the current locale or check if the locale is a given value:

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

$locale = App::currentLocale();

if (App::isLocale('en')) {
    //
}
```

Defining Translation Strings

Using Short Keys

Typically, translation strings are stored in files within the `resources/lang` directory. Within this directory there should be a subdirectory for each language supported by your application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files return an array of keyed strings. For example:

```
<?php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application!',
];
```

> NOTE For languages that differ by territory, you should name the language directories according to the ISO 15897. For example, "en_GB" should be used for British English rather than "en-gb".

Using Translation Strings As Keys

For applications with a large number of translatable strings, defining every string with a "short key" can become confusing when referencing the keys in your views and it is cumbersome to continually invent keys for every translation string supported by your application.

For this reason, Laravel also provides support for defining translation strings using the "default" translation of the string as the key. Translation files that use translation strings as keys are stored as JSON files in the `resources/lang` directory. For example, if your application has a Spanish translation, you should create a `resources/lang/es.json` file:

```
{
    "I love programming.": "Me encanta programar."
}
```

Key / File Conflicts

You should not define translation string keys that conflict with other translation filenames. For example, translating `__('Action')` for the "NL" locale while a `nl/action.php` file exists but a `nl.json` file does not exist will result in the translator returning the contents of `nl/action.php`.

Retrieving Translation Strings

You may retrieve translation strings from your language files using the __ helper function. If you are using "short keys" to define your translation strings, you should pass the file that contains the key and the key itself to the __ function using "dot" syntax. For example, let's retrieve the `welcome` translation string from the `resources/lang/en/messages.php` language file:

```
echo __('messages.welcome');
```

If the specified translation string does not exist, the __ function will return the translation string key. So, using the example above, the __ function would return `messages.welcome` if the translation string does not exist.

If you are using your default translation strings as your translation keys, you should pass the default translation of your string to the __ function;

```
echo __('I love programming.');
```

Again, if the translation string does not exist, the __ function will return the translation string key that it was given.

If you are using the Blade templating engine, you may use the `{{ }}` echo syntax to display the translation string:

```
{{ __('messages.welcome') }}
```

Replacing Parameters In Translation Strings

If you wish, you may define placeholders in your translation strings. All placeholders are prefixed with a `:`. For example, you may define a welcome message with a placeholder name:

```
'welcome' => 'Welcome, :name',
```

To replace the placeholders when retrieving a translation string, you may pass an array of replacements as the second argument to the __ function:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

If your placeholder contains all capital letters, or only has its first letter capitalized, the translated value will be capitalized accordingly:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization; however, Laravel can help you translate strings differently based on pluralization rules that you define. Using a `|` character, you may distinguish singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

Of course, pluralization is also supported when using translation strings as keys:

```
{
    "There is one apple|There are many apples": "Hay una manzana|Hay muchas manzanas"
}
```

You may even create more complex pluralization rules which specify translation strings for multiple ranges of values:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

After defining a translation string that has pluralization options, you may use the `trans_choice` function to retrieve the line for a given "count". In this example, since the count is greater than one, the plural form of the translation string is returned:

```
echo trans_choice('messages.apples', 10);
```

You may also define placeholder attributes in pluralization strings. These placeholders may be replaced by passing an array as the third argument to the `trans_choice` function:

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',
```

```
echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

If you would like to display the integer value that was passed to the `trans_choice` function, you may use the built-in `:count` placeholder:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

Overriding Package Language Files

Some packages may ship with their own language files. Instead of changing the package's core files to tweak these lines, you may override them by placing files in the `resources/lang/vendor/{package}/{locale}` directory.

So, for example, if you need to override the English translation strings in `messages.php` for a package named `skyrim/hearthfire`, you should place a language file at: `resources/lang/vendor/hearthfire/en/messages.php`. Within this file, you should only define the translation strings you wish to override. Any translation strings you don't override will still be loaded from the package's original language files.

Digging Deeper

Mail

- Introduction
  - Configuration
  - Driver Prerequisites
- Generating Mailables
- Writing Mailables
  - Configuring The Sender
  - Configuring The View
  - View Data
  - Attachments
  - Inline Attachments
  - Customizing The SwiftMailer Message
- Markdown Mailables
  - Generating Markdown Mailables
  - Writing Markdown Messages
  - Customizing The Components
- Sending Mail
  - Queueing Mail
- Rendering Mailables
  - Previewing Mailables In The Browser
- Localizing Mailables
- Testing Mailables
- Mail & Local Development
- Events

Introduction

Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular SwiftMailer library. Laravel and SwiftMailer provide drivers for sending email via SMTP, Mailgun, Postmark, Amazon SES, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Configuration

Laravel's email services may be configured via your application's `config/mail.php` configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.

Within your `mail` configuration file, you will find a `mailers` configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the `default` configuration value determines which mailer will be used by default when your application needs to send an email message.

Driver / Transport Prerequisites

The API based drivers such as Mailgun and Postmark are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers. All of the API based drivers require the Guzzle HTTP library, which may be installed via the Composer package manager:

```
composer require guzzlehttp/guzzle
```

Mailgun Driver

To use the Mailgun driver, first install the Guzzle HTTP library. Then, set the `default` option in your

`config/mail.php` configuration file to `mailgun`. Next, verify that your `config/services.php` configuration file contains the following options:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
],
```

If you are not using the United States Mailgun region, you may define your region's endpoint in the `services` configuration file:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
],
```

Postmark Driver

To use the Postmark driver, install Postmark's SwiftMailer transport via Composer:

```
composer require wildbit/swiftmailer-postmark
```

Next, install the Guzzle HTTP library and set the `default` option in your `config/mail.php` configuration file to `postmark`. Finally, verify that your `config/services.php` configuration file contains the following options:

```
'postmark' => [
    'token' => env('POSTMARK_TOKEN'),
],
```

SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library via the Composer package manager:

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

Next, set the `default` option in your `config/mail.php` configuration file to `ses` and verify that your `config/services.php` configuration file contains the following options:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

If you would like to define additional options that Laravel should pass to the AWS SDK's `SendRawEmail` method when sending an email, you may define an `options` array within your `ses` configuration:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'Tags' => [
            ['Name' => 'foo', 'Value' => 'bar'],
        ],
    ],
],
```

Generating Mailables

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the `app/Mail` directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped
```

Writing Mailables

Once you have generated a mailable class, open it up so we can explore its contents. First, note that all of a mailable class' configuration is done in the `build` method. Within this method, you may call various methods such as `from`, `subject`, `view`, and `attach` to configure the email's presentation and delivery.

Configuring The Sender

Using The `from` Method

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may use the `from` method within your mailable class' `build` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
                ->view('emails.orders.shipped');
}
```

Using A Global `from` Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to call the `from` method in each mailable class you generate. Instead, you may specify a global "from" address in your `config/mail.php` configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

In addition, you may define a global "reply_to" address within your `config/mail.php` configuration file:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

Configuring The View

Within a mailable class' `build` method, you may use the `view` method to specify which template should be used when rendering the email's contents. Since each email typically uses a Blade template to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
```

> TIP You may wish to create a `resources/views/emails` directory to house all of your email templates; however, you are free to place them wherever you wish within your `resources/views` directory.

Plain Text Emails

If you would like to define a plain-text version of your email, you may use the `text` method. Like the `view` method, the `text` method accepts a template name which will be used to render the contents of the email. You are free to define both an HTML and plain-text version of your message:

```
/**
 * Build the message.
```

```
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->text('emails.orders.shipped_plain');
}
```

View Data

Via Public Properties

Typically, you will want to pass some data to your view that you can utilize when rendering the email's HTML. There are two ways you may make data available to your view. First, any public property defined on your mailable class will automatically be made available to the view. So, for example, you may pass data into your mailable class' constructor and set that data to public properties defined on the class:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;

    /**
     * Create a new message instance.
     *
     * @param  \App\Models\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped');
    }
}
```

Once the data has been set to a public property, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $order->price }}
</div>
```

Via The `with` Method:

If you would like to customize the format of your email's data before it is sent to the template, you may manually pass your data to the view via the `with` method. Typically, you will still pass data via the mailable class' constructor; however, you should set this data to `protected` or `private` properties so the data is not automatically made available to the template. Then, when calling the `with` method, pass an array of data that

you wish to make available to the template:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    protected $order;

    /**
     * Create a new message instance.
     *
     * @param  \App\Models\Order $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
                    ->with([
                        'orderName' => $this->order->name,
                        'orderPrice' => $this->order->price,
                    ]);
    }
}
```

Once the data has been passed to the `with` method, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $orderPrice }}
</div>
```

Attachments

To add attachments to an email, use the `attach` method within the mailable class' `build` method. The `attach` method accepts the full path to the file as its first argument:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attach('/path/to/file');
}
```

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Build the message.
```

```
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
                ]);
}
```

Attaching Files from Disk

If you have stored a file on one of your filesystem disks, you may attach it to the email using the `attachFromStorage` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
                ->attachFromStorage('/path/to/file');
}
```

If necessary, you may specify the file's attachment name and additional options using the second and third arguments to the `attachFromStorage` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
                ->attachFromStorage('/path/to/file', 'name.pdf', [
                    'mime' => 'application/pdf'
                ]);
}
```

The `attachFromStorageDisk` method may be used if you need to specify a storage disk other than your default disk:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
                ->attachFromStorageDisk('s3', '/path/to/file');
}
```

Raw Data Attachments

The `attachData` method may be used to attach a raw string of bytes as an attachment. For example, you might use this method if you have generated a PDF in memory and want to attach it to the email without writing it to disk. The `attachData` method accepts the raw data bytes as its first argument, the name of the file as its second argument, and an array of options as its third argument:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                 ->attachData($this->pdf, 'name.pdf', [
```

```
                    'mime' => 'application/pdf',
                ]);
}
```

Inline Attachments

Embedding inline images into your emails is typically cumbersome; however, Laravel provides a convenient way to attach images to your emails. To embed an inline image, use the `embed` method on the `$message` variable within your email template. Laravel automatically makes the `$message` variable available to all of your email templates, so you don't need to worry about passing it in manually:

```
<body>
    Here is an image:

    <img src="{{ $message->embed($pathToImage) }}">
</body>
```

> NOTE The `$message` variable is not available in plain-text message templates since plain-text messages do not utilize inline attachments.

Embedding Raw Data Attachments

If you already have a raw image data string you wish to embed into an email template, you may call the `embedData` method on the `$message` variable. When calling the `embedData` method, you will need to provide a filename that should be assigned to the embedded image:

```
<body>
    Here is an image from raw data:

    <img src="{{ $message->embedData($data, 'example-image.jpg') }}">
</body>
```

Customizing The SwiftMailer Message

The `withSwiftMessage` method of the `Mailable` base class allows you to register a closure which will be invoked with the SwiftMailer message instance before sending the message. This gives you an opportunity to deeply customize the message before it is delivered:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    $this->view('emails.orders.shipped');

    $this->withSwiftMessage(function ($message) {
        $message->getHeaders()->addTextHeader(
            'Custom-Header', 'Header Value'
        );
    });
}
```

Markdown Mailables

Markdown mailable messages allow you to take advantage of the pre-built templates and components of mail notifications in your mailables. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating Markdown Mailables

To generate a mailable with a corresponding Markdown template, you may use the `--markdown` option of the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

Then, when configuring the mailable within its `build` method, call the `markdown` method instead of the `view` method. The `markdown` method accepts the name of the Markdown template and an optional array of data to make available to the template:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
                ->markdown('emails.orders.shipped', [
                    'url' => $this->orderUrl,
                ]);
}
```

Writing Markdown Messages

Markdown mailables use a combination of Blade components and Markdown syntax which allow you to easily construct mail messages while leveraging Laravel's pre-built email UI components:

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

> TIP Do not use excess indentation when writing Markdown emails. Per Markdown standards, Markdown parsers will render indented content as code blocks.

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `success`, and `error`. You may add as many button components to a message as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View Order
@endcomponent
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the message. This allows you to draw attention to a given block of text:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
```

```
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Customizing The Components

You may export all of the Markdown mail components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

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

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing The CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be converted to inline CSS styles within the HTML representations of your Markdown mail messages.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of your application's `config/mail.php` configuration file to match the name of your new theme.

To customize the theme for an individual mailable, you may set the `$theme` property of the mailable class to the name of the theme that should be used when sending that mailable.

Sending Mail

To send a message, use the `to` method on the `Mail` facade. The `to` method accepts an email address, a user instance, or a collection of users. If you pass an object or collection of objects, the mailer will automatically use their `email` and `name` properties when determining the email's recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $order = Order::findOrFail($request->order_id);

        // Ship the order...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}
```

You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients by chaining their respective methods together:

```php
Mail::to($request->user())
    ->cc($moreUsers)
```

```
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

Looping Over Recipients

Occasionally, you may need to send a mailable to a list of recipients by iterating over an array of recipients / email addresses. However, since the `to` method appends email addresses to the mailable's list of recipients, each iteration through the loop will send another email to every previous recipient. Therefore, you should always re-create the mailable instance for each recipient:

```
foreach (['taylor@example.com', 'dries@example.com'] as $recipient) {
    Mail::to($recipient)->send(new OrderShipped($order));
}
```

Sending Mail Via A Specific Mailer

By default, Laravel will send email using the mailer configured as the `default` mailer in your application's `mail` configuration file. However, you may use the `mailer` method to send a message using a specific mailer configuration:

```
Mail::mailer('postmark')
        ->to($request->user())
        ->send(new OrderShipped($order));
```

Queueing Mail

Queueing A Mail Message

Since sending email messages can negatively impact the response time of your application, many developers choose to queue email messages for background sending. Laravel makes this easy using its built-in unified queue API. To queue a mail message, use the `queue` method on the `Mail` facade after specifying the message's recipients:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

This method will automatically take care of pushing a job onto the queue so the message is sent in the background. You will need to configure your queues before using this feature.

Delayed Message Queueing

If you wish to delay the delivery of a queued email message, you may use the `later` method. As its first argument, the `later` method accepts a `DateTime` instance indicating when the message should be sent:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

Pushing To Specific Queues

Since all mailable classes generated using the `make:mail` command make use of the `Illuminate\Bus\Queueable` trait, you may call the `onQueue` and `onConnection` methods on any mailable class instance, allowing you to specify the connection and queue name for the message:

```
$message = (new OrderShipped($order))
                ->onConnection('sqs')
                ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

Queueing By Default

If you have mailable classes that you want to always be queued, you may implement the `ShouldQueue` contract on the class. Now, even if you call the `send` method when mailing, the mailable will still be queued since it implements the contract:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    //
}
```

Queued Mailables & Database Transactions

When queued mailables are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your mailable depends on these models, unexpected errors can occur when the job that sends the queued mailable is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued mailable should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the mailable class:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    public $afterCommit = true;
}
```

> TIP To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Rendering Mailables

Sometimes you may wish to capture the HTML content of a mailable without sending it. To accomplish this, you may call the `render` method of the mailable. This method will return the evaluated HTML content of the mailable as a string:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;

$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))->render();
```

Previewing Mailables In The Browser

When designing a mailable's template, it is convenient to quickly preview the rendered mailable in your browser like a typical Blade template. For this reason, Laravel allows you to return any mailable directly from a route closure or controller. When a mailable is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
Route::get('/mailable', function () {
    $invoice = App\Models\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

> NOTE Inline attachments will not be rendered when a mailable is previewed in your browser. To preview these mailables, you should send them to an email testing application such as MailHog or HELO.

Localizing Mailables

Laravel allows you to send mailables in a locale other than the request's current locale, and will even remember this locale if the mail is queued.

To accomplish this, the `Mail` facade offers a `locale` method to set the desired language. The application will change into this locale when the mailable's template is being evaluated and then revert back to the previous locale when evaluation is complete:

```
Mail::to($request->user())->locale('es')->send(
    new OrderShipped($order)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on one or more of your models, you may instruct Laravel to use this stored locale when sending mail:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     *
     * @return string
     */
    public function preferredLocale()
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending mailables and notifications to the model. Therefore, there is no need to call the `locale` method when using this interface:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

Testing Mailables

Laravel provides several convenient methods for testing that your mailables contain the content that you expect. These methods are: `assertSeeInHtml`, `assertDontSeeInHtml`, `assertSeeInText`, and `assertDontSeeInText`.

As you might expect, the "HTML" assertions assert that the HTML version of your mailable contains a given string, while the "text" assertions assert that the plain-text version of your mailable contains a given string:

```
use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content()
{
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInText('Invoice Paid');
}
```

Testing Mailable Sending

We suggest testing the content of your mailables separately from your tests that assert that a given mailable was "sent" to a specific user. To learn how to test that mailables were sent, check out our documentation on the Mail fake.

Mail & Local Development

When developing an application that sends email, you probably don't want to actually send emails to live email addresses. Laravel provides several ways to "disable" the actual sending of emails during local development.

Log Driver

Instead of sending your emails, the `log` mail driver will write all email messages to your log files for inspection. Typically, this driver would only be used during local development. For more information on configuring your application per environment, check out the configuration documentation.

HELO / Mailtrap / MailHog

Finally, you may use a service like HELO or Mailtrap and the `smtp` driver to send your email messages to a "dummy" mailbox where you may view them in a true email client. This approach has the benefit of allowing you to actually inspect the final emails in Mailtrap's message viewer.

If you are using Laravel Sail, you may preview your messages using MailHog. When Sail is running, you may access the MailHog interface at: `http://localhost:8025`.

Events

Laravel fires two events during the process of sending mail messages. The `MessageSending` event is fired prior to a message being sent, while the `MessageSent` event is fired after a message has been sent. Remember, these events are fired when the mail is being *sent*, not when it is queued. You may register event listeners for this event in your `App\Providers\EventServiceProvider` service provider:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSendingMessage',
    ],
    'Illuminate\Mail\Events\MessageSent' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

Digging Deeper

Notifications

- Introduction
- Generating Notifications
- Sending Notifications
  - Using The Notifiable Trait
  - Using The Notification Facade
  - Specifying Delivery Channels
  - Queueing Notifications
  - On-Demand Notifications
- Mail Notifications
  - Formatting Mail Messages
  - Customizing The Sender
  - Customizing The Recipient
  - Customizing The Subject
  - Customizing The Mailer
  - Customizing The Templates
  - Attachments
  - Previewing Mail Notifications
- Markdown Mail Notifications
  - Generating The Message
  - Writing The Message
  - Customizing The Components
- Database Notifications
  - Prerequisites
  - Formatting Database Notifications
  - Accessing The Notifications
  - Marking Notifications As Read
- Broadcast Notifications
  - Prerequisites
  - Formatting Broadcast Notifications
  - Listening For Notifications
- SMS Notifications
  - Prerequisites
  - Formatting SMS Notifications
  - Formatting Shortcode Notifications
  - Customizing The "From" Number
  - Routing SMS Notifications
- Slack Notifications
  - Prerequisites
  - Formatting Slack Notifications
  - Slack Attachments
  - Routing Slack Notifications
- Localizing Notifications
- Notification Events
- Custom Channels

Introduction

In addition to support for sending email, Laravel provides support for sending notifications across a variety of delivery channels, including email, SMS (via Vonage, formerly known as Nexmo), and Slack. In addition, a variety of community built notification channels have been created to send notification over dozens of different channels! Notifications may also be stored in a database so they may be displayed in your web interface.

Typically, notifications should be short, informational messages that notify users of something that occurred in your application. For example, if you are writing a billing application, you might send an "Invoice Paid" notification to your users via the email and SMS channels.

Generating Notifications

In Laravel, each notification is represented by a single class that is typically stored in the `app/Notifications` directory. Don't worry if you don't see this directory in your application - it will be created for you when you run the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid
```

This command will place a fresh notification class in your `app/Notifications` directory. Each notification class contains a `via` method and a variable number of message building methods, such as `toMail` or `toDatabase`, that convert the notification to a message tailored for that particular channel.

Sending Notifications

Using The Notifiable Trait

Notifications may be sent in two ways: using the `notify` method of the `Notifiable` trait or using the `Notification` facade. The `Notifiable` trait is included on your application's `App\Models\User` model by default:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

The `notify` method that is provided by this trait expects to receive a notification instance:

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```

> TIP Remember, you may use the `Notifiable` trait on any of your models. You are not limited to only including it on your `User` model.

Using The Notification Facade

Alternatively, you may send notifications via the `Notification` facade. This approach is useful when you need to send a notification to multiple notifiable entities such as a collection of users. To send notifications using the facade, pass all of the notifiable entities and the notification instance to the `send` method:

```
use Illuminate\Support\Facades\Notification;

Notification::send($users, new InvoicePaid($invoice));
```

Specifying Delivery Channels

Every notification class has a `via` method that determines on which channels the notification will be delivered. Notifications may be sent on the `mail`, `database`, `broadcast`, `nexmo`, and `slack` channels.

> TIP If you would like to use other delivery channels such as Telegram or Pusher, check out the community driven Laravel Notification Channels website.

The `via` method receives a `$notifiable` instance, which will be an instance of the class to which the notification is being sent. You may use `$notifiable` to determine which channels the notification should be delivered on:

```
/**
 * Get the notification's delivery channels.
 *
 * @param  mixed  $notifiable
```

```
 * @return array
 */
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

Queueing Notifications

NOTE Before queueing notifications you should configure your queue and start a worker.

Sending notifications can take time, especially if the channel needs to make an external API call to deliver the notification. To speed up your application's response time, let your notification be queued by adding the `ShouldQueue` interface and `Queueable` trait to your class. The interface and trait are already imported for all notifications generated using the `make:notification` command, so you may immediately add them to your notification class:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

Once the `ShouldQueue` interface has been added to your notification, you may send the notification like normal. Laravel will detect the `ShouldQueue` interface on the class and automatically queue the delivery of the notification:

```
$user->notify(new InvoicePaid($invoice));
```

If you would like to delay the delivery of the notification, you may chain the `delay` method onto your notification instantiation:

```
$delay = now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($delay));
```

You may pass an array to the `delay` method to specify the delay amount for specific channels:

```
$user->notify((new InvoicePaid($invoice))->delay([
    'mail' => now()->addMinutes(5),
    'sms' => now()->addMinutes(10),
]));
```

When queueing notifications, a queued job will be created for each recipient and channel combination. For example, six jobs will be dispatched to the queue if your notification has three recipients and two channels.

Customizing The Notification Queue Connection

By default, queued notifications will be queued using your application's default queue connection. If you would like to specify a different connection that should be used for a particular notification, you may define a `$connection` property on the notification class:

```
/**
 * The name of the queue connection to use when queueing the notification.
 *
 * @var string
 */
public $connection = 'redis';
```

Customizing Notification Channel Queues

If you would like to specify a specific queue that should be used for each notification channel supported by the notification, you may define a `viaQueues` method on your notification. This method should return an array of channel name / queue name pairs:

```
/**
 * Determine which queues should be used for each notification channel.
 *
 * @return array
 */
public function viaQueues()
{
    return [
        'mail' => 'mail-queue',
        'slack' => 'slack-queue',
    ];
}
```

Queued Notifications & Database Transactions

When queued notifications are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your notification depends on these models, unexpected errors can occur when the job that sends the queued notification is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued notification should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the notification class:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    public $afterCommit = true;
}
```

> TIP To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

On-Demand Notifications

Sometimes you may need to send a notification to someone who is not stored as a "user" of your application. Using the `Notification` facade's `route` method, you may specify ad-hoc notification routing information before sending the notification:

```
Notification::route('mail', 'taylor@example.com')
            ->route('nexmo', '5555555555')
            ->route('slack', 'https://hooks.slack.com/services/...')
            ->notify(new InvoicePaid($invoice));
```

Mail Notifications

Formatting Mail Messages

If a notification supports being sent as an email, you should define a `toMail` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\MailMessage` instance.

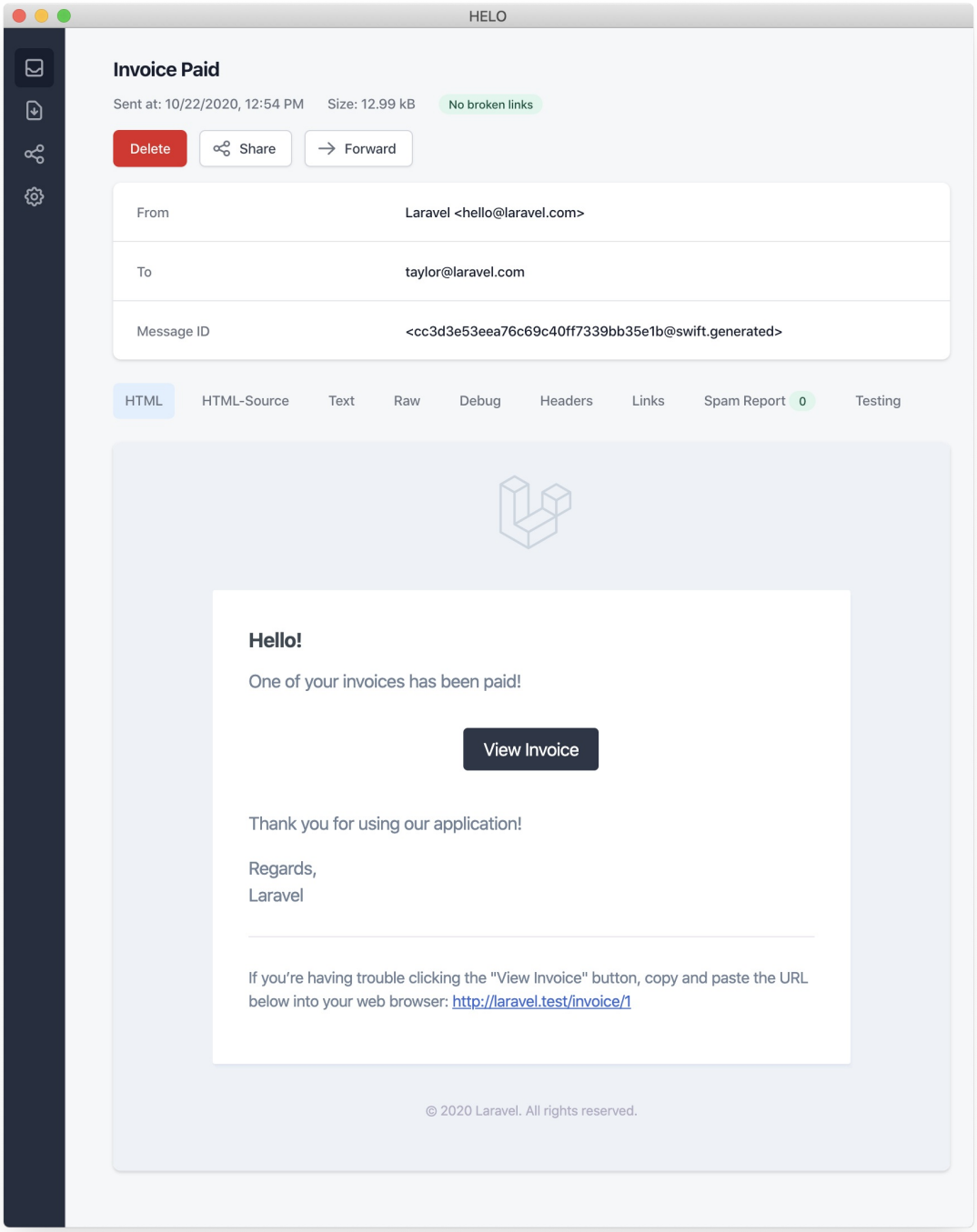
The `MailMessage` class contains a few simple methods to help you build transactional email messages. Mail messages may contain lines of text as well as a "call to action". Let's take a look at an example `toMail` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->greeting('Hello!')
                ->line('One of your invoices has been paid!')
                ->action('View Invoice', $url)
                ->line('Thank you for using our application!');
}
```

TIP Note we are using `$this->invoice->id` in our `toMail` method. You may pass any data your notification needs to generate its message into the notification's constructor.

In this example, we register a greeting, a line of text, a call to action, and then another line of text. These methods provided by the `MailMessage` object make it simple and fast to format small transactional emails. The mail channel will then translate the message components into a beautiful, responsive HTML email template with a plain-text counterpart. Here is an example of an email generated by the `mail` channel:



TIP When sending mail notifications, be sure to set the `name` configuration option in your `config/app.php` configuration file. This value will be used in the header and footer of your mail notification messages.

Other Mail Notification Formatting Options

Instead of defining the "lines" of text in the notification class, you may use the `view` method to specify a custom template that should be used to render the notification email:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
```

```
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        'emails.name', ['invoice' => $this->invoice]
    );
}
```

You may specify a plain-text view for the mail message by passing the view name as the second element of an array that is given to the `view` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        ['emails.name.html', 'emails.name.plain'],
        ['invoice' => $this->invoice]
    );
}
```

In addition, you may return a full mailable object from the `toMail` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email);
}
```

Error Messages

Some notifications inform users of errors, such as a failed invoice payment. You may indicate that a mail message is regarding an error by calling the `error` method when building your message. When using the `error` method on a mail message, the call to action button will be red instead of black:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Message
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->error()
                ->subject('Notification Subject')
                ->line('...');
}
```

Customizing The Sender

By default, the email's sender / from address is defined in the `config/mail.php` configuration file. However, you may specify the from address for a specific notification using the `from` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
```

```
{
    return (new MailMessage)
                ->from('barrett@example.com', 'Barrett Blair')
                ->line('...');
}
```

Customizing The Recipient

When sending notifications via the `mail` channel, the notification system will automatically look for an `email` property on your notifiable entity. You may customize which email address is used to deliver the notification by defining a `routeNotificationForMail` method on the notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return array|string
     */
    public function routeNotificationForMail($notification)
    {
        // Return email address only...
        return $this->email_address;

        // Return email address and name...
        return [$this->email_address => $this->name];
    }
}
```

Customizing The Subject

By default, the email's subject is the class name of the notification formatted to "Title Case". So, if your notification class is named `InvoicePaid`, the email's subject will be `Invoice Paid`. If you would like to specify an different subject for the message, you may call the `subject` method when building your message:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->subject('Notification Subject')
                ->line('...');
}
```

Customizing The Mailer

By default, the email notification will be sent using the default mailer defined in the `config/mail.php` configuration file. However, you may specify a different mailer at runtime by calling the `mailer` method when building your message:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
```

```
                ->mailer('postmark')
                ->line('...');
}
```

Customizing The Templates

You can modify the HTML and plain-text template used by mail notifications by publishing the notification package's resources. After running this command, the mail notification templates will be located in the `resources/views/vendor/notifications` directory:

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

Attachments

To add attachments to an email notification, use the `attach` method while building your message. The `attach` method accepts the absolute path to the file as its first argument:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attach('/path/to/file');
}
```

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
                ]);
}
```

Unlike attaching files in mailable objects, you may not attach a file directly from a storage disk using `attachFromStorage`. You should rather use the `attach` method with an absolute path to the file on the storage disk. Alternatively, you could return a mailable from the `toMail` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email)
                ->attachFromStorage('/path/to/file');
}
```

Raw Data Attachments

The `attachData` method may be used to attach a raw string of bytes as an attachment. When calling the

`attachData` method, you should provide the filename that should be assigned to the attachment:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attachData($this->pdf, 'name.pdf', [
                    'mime' => 'application/pdf',
                ]);
}
```

Previewing Mail Notifications

When designing a mail notification template, it is convenient to quickly preview the rendered mail message in your browser like a typical Blade template. For this reason, Laravel allows you to return any mail message generated by a mail notification directly from a route closure or controller. When a `MailMessage` is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
use App\Models\Invoice;
use App\Notifications\InvoicePaid;

Route::get('/notification', function () {
    $invoice = Invoice::find(1);

    return (new InvoicePaid($invoice))
                ->toMail($invoice->user);
});
```

Markdown Mail Notifications

Markdown mail notifications allow you to take advantage of the pre-built templates of mail notifications, while giving you more freedom to write longer, customized messages. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating The Message

To generate a notification with a corresponding Markdown template, you may use the `--markdown` option of the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

Like all other mail notifications, notifications that use Markdown templates should define a `toMail` method on their notification class. However, instead of using the `line` and `action` methods to construct the notification, use the `markdown` method to specify the name of the Markdown template that should be used. An array of data you wish to make available to the template may be passed as the method's second argument:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->subject('Invoice Paid')
                ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Writing The Message

Markdown mail notifications use a combination of Blade components and Markdown syntax which allow you to easily construct notifications while leveraging Laravel's pre-crafted notification components:

```
@component('mail::message')
# Invoice Paid

Your invoice has been paid!

@component('mail::button', ['url' => $url])
View Invoice
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `green`, and `red`. You may add as many button components to a notification as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
View Invoice
@endcomponent
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the notification. This allows you to draw attention to a given block of text:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Customizing The Components

You may export all of the Markdown notification components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

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

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing The CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be in-lined within the HTML representations of your Markdown notifications.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of the `mail` configuration file to match the name of your new theme.

To customize the theme for an individual notification, you may call the `theme` method while building the notification's mail message. The `theme` method accepts the name of the theme that should be used when sending the notification:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->theme('invoice')
                ->subject('Invoice Paid')
                ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Database Notifications

Prerequisites

The `database` notification channel stores the notification information in a database table. This table will contain information such as the notification type as well as a JSON data structure that describes the notification.

You can query the table to display the notifications in your application's user interface. But, before you can do that, you will need to create a database table to hold your notifications. You may use the `notifications:table` command to generate a migration with the proper table schema:

```
php artisan notifications:table
```

```
php artisan migrate
```

Formatting Database Notifications

If a notification supports being stored in a database table, you should define a `toDatabase` or `toArray` method on the notification class. This method will receive a `$notifiable` entity and should return a plain PHP array. The returned array will be encoded as JSON and stored in the `data` column of your `notifications` table. Let's take a look at an example `toArray` method:

```
/**
 * Get the array representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

`toDatabase` Vs. `toArray`

The `toArray` method is also used by the `broadcast` channel to determine which data to broadcast to your JavaScript powered frontend. If you would like to have two different array representations for the `database` and `broadcast` channels, you should define a `toDatabase` method instead of a `toArray` method.

Accessing The Notifications

Once notifications are stored in the database, you need a convenient way to access them from your notifiable entities. The `Illuminate\Notifications\Notifiable` trait, which is included on Laravel's default `App\Models\User` model, includes a `notifications` Eloquent relationship that returns the notifications for the entity. To fetch notifications, you may access this method like any other Eloquent relationship. By default, notifications will be sorted by the `created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

If you want to retrieve only the "unread" notifications, you may use the `unreadNotifications` relationship. Again, these notifications will be sorted by the `created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

> TIP To access your notifications from your JavaScript client, you should define a notification controller for your application which returns the notifications for a notifiable entity, such as the current user. You may then make an HTTP request to that controller's URL from your JavaScript client.

Marking Notifications As Read

Typically, you will want to mark a notification as "read" when a user views it. The `Illuminate\Notifications\Notifiable` trait provides a `markAsRead` method, which updates the `read_at` column on the notification's database record:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

However, instead of looping through each notification, you may use the `markAsRead` method directly on a collection of notifications:

```
$user->unreadNotifications->markAsRead();
```

You may also use a mass-update query to mark all of the notifications as read without retrieving them from the database:

```
$user = App\Models\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

You may `delete` the notifications to remove them from the table entirely:

```
$user->notifications()->delete();
```

Broadcast Notifications

Prerequisites

Before broadcasting notifications, you should configure and be familiar with Laravel's event broadcasting services. Event broadcasting provides a way to react to server-side Laravel events from your JavaScript powered frontend.

Formatting Broadcast Notifications

The `broadcast` channel broadcasts notifications using Laravel's event broadcasting services, allowing your

JavaScript powered frontend to catch notifications in realtime. If a notification supports broadcasting, you can define a `toBroadcast` method on the notification class. This method will receive a `$notifiable` entity and should return a `BroadcastMessage` instance. If the `toBroadcast` method does not exist, the `toArray` method will be used to gather the data that should be broadcast. The returned data will be encoded as JSON and broadcast to your JavaScript powered frontend. Let's take a look at an example `toBroadcast` method:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the broadcastable representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

Broadcast Queue Configuration

All broadcast notifications are queued for broadcasting. If you would like to configure the queue connection or queue name that is used to queue the broadcast operation, you may use the `onConnection` and `onQueue` methods of the `BroadcastMessage`:

```
return (new BroadcastMessage($data))
                ->onConnection('sqs')
                ->onQueue('broadcasts');
```

Customizing The Notification Type

In addition to the data you specify, all broadcast notifications also have a `type` field containing the full class name of the notification. If you would like to customize the notification `type`, you may define a `broadcastType` method on the notification class:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the type of the notification being broadcast.
 *
 * @return string
 */
public function broadcastType()
{
    return 'broadcast.message';
}
```

Listening For Notifications

Notifications will broadcast on a private channel formatted using a `{notifiable}.{id}` convention. So, if you are sending a notification to an `App\Models\User` instance with an ID of `1`, the notification will be broadcast on the `App.Models.User.1` private channel. When using Laravel Echo, you may easily listen for notifications on a channel using the `notification` method:

```
Echo.private('App.Models.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
```

Customizing The Notification Channel

If you would like to customize which channel that an entity's broadcast notifications are broadcast on, you may define a `receivesBroadcastNotificationsOn` method on the notifiable entity:

```
<?php
```

```
namespace App\Models;

use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The channels the user receives notification broadcasts on.
     *
     * @return string
     */
    public function receivesBroadcastNotificationsOn()
    {
        return 'users.'.$this->id;
    }
}
```

SMS Notifications

Prerequisites

Sending SMS notifications in Laravel is powered by Vonage (formerly known as Nexmo). Before you can send notifications via Vonage, you need to install the `laravel/nexmo-notification-channel` and `nexmo/laravel` Composer packages

```
composer require laravel/nexmo-notification-channel nexmo/laravel
```

The `nexmo/laravel` package includes its own configuration file. However, you are not required to export this configuration file to your own application. You can simply use the `NEXMO_KEY` and `NEXMO_SECRET` environment variables to set your Vonage public and secret key.

Next, you will need to add a `nexmo` configuration entry to your `config/services.php` configuration file. You may copy the example configuration below to get started:

```
'nexmo' => [
    'sms_from' => '15556666666',
],
```

The `sms_from` option is the phone number that your SMS messages will be sent from. You should generate a phone number for your application in the Vonage control panel.

Formatting SMS Notifications

If a notification supports being sent as an SMS, you should define a `toNexmo` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\NexmoMessage` instance:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your SMS message content');
}
```

Unicode Content

If your SMS message will contain unicode characters, you should call the `unicode` method when constructing the `NexmoMessage` instance:

```
/**
```

```
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your unicode message')
                ->unicode();
}
```

Formatting Shortcode Notifications

Laravel also supports sending shortcode notifications, which are pre-defined message templates in your Vonage account. To send a shortcode SMS notification, you should define a `toShortcode` method on your notification class. From within this method, you may return an array specifying the type of notification (`alert`, `2fa`, or `marketing`) as well as the custom values that will populate the template:

```
/**
 * Get the Vonage / Shortcode representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return array
 */
public function toShortcode($notifiable)
{
    return [
        'type' => 'alert',
        'custom' => [
            'code' => 'ABC123',
        ];
    ];
}
```

> TIP Like routing SMS Notifications, you should implement the `routeNotificationForShortcode` method on your notifiable model.

Customizing The "From" Number

If you would like to send some notifications from a phone number that is different from the phone number specified in your `config/services.php` file, you may call the `from` method on a `NexmoMessage` instance:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your SMS message content')
                ->from('15554443333');
}
```

Routing SMS Notifications

To route Vonage notifications to the proper phone number, define a `routeNotificationForNexmo` method on your notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
```

```
    /**
     * Route notifications for the Nexmo channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return string
     */
    public function routeNotificationForNexmo($notification)
    {
        return $this->phone_number;
    }
}
```

Slack Notifications

Prerequisites

Before you can send notifications via Slack, you must install the Slack notification channel via Composer:

```
composer require laravel/slack-notification-channel
```

You will also need to configure an "Incoming Webhook" integration for your Slack team. This integration will provide you with a URL you may use when routing Slack notifications.

Formatting Slack Notifications

If a notification supports being sent as a Slack message, you should define a `toSlack` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\SlackMessage` instance. Slack messages may contain text content as well as an "attachment" that formats additional text or an array of fields. Let's take a look at a basic `toSlack` example:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Message\SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->content('One of your invoices has been paid!');
}
```

Customizing The Sender & Recipient

You may use the `from` and `to` methods to customize the sender and recipient. The `from` method accepts a username and emoji identifier, while the `to` method accepts a channel or username:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->from('Ghost', ':ghost:')
                ->to('#bots')
                ->content('This will be sent to #bots');
}
```

You may also use an image as your from "logo" instead of an emoji:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
```

```
    return (new SlackMessage)
                ->from('Laravel')
                ->image('https://laravel.com/img/favicon/favicon.ico')
                ->content('This will display the Laravel logo next to the message');
}
```

Slack Attachments

You may also add "attachments" to Slack messages. Attachments provide richer formatting options than simple text messages. In this example, we will send an error notification about an exception that occurred in an application, including a link to view more details about the exception:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
                ->error()
                ->content('Whoops! Something went wrong.')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Exception: File Not Found', $url)
                               ->content('File [background.jpg] was not found.');
                });
}
```

Attachments also allow you to specify an array of data that should be presented to the user. The given data will be presented in a table-style format for easy reading:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/invoices/'.$this->invoice->id);

    return (new SlackMessage)
                ->success()
                ->content('One of your invoices has been paid!')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Invoice 1322', $url)
                               ->fields([
                                    'Title' => 'Server Expenses',
                                    'Amount' => '$1,234',
                                    'Via' => 'American Express',
                                    'Was Overdue' => ':-1:',
                                ]);
                });
}
```

Markdown Attachment Content

If some of your attachment fields contain Markdown, you may use the `markdown` method to instruct Slack to parse and display the given attachment fields as Markdown formatted text. The values accepted by this method are: `pretext`, `text`, and / or `fields`. For more information about Slack attachment formatting, check out the Slack API documentation:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);
```

```
    return (new SlackMessage)
                ->error()
                ->content('Whoops! Something went wrong.')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Exception: File Not Found', $url)
                               ->content('File [background.jpg] was *not found*.')
                               ->markdown(['text']);
                });
}
```

Routing Slack Notifications

To route Slack notifications to the proper Slack team and channel, define a `routeNotificationForSlack` method on your notifiable entity. This should return the webhook URL to which the notification should be delivered. Webhook URLs may be generated by adding an "Incoming Webhook" service to your Slack team:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return string
     */
    public function routeNotificationForSlack($notification)
    {
        return 'https://hooks.slack.com/services/...';
    }
}
```

Localizing Notifications

Laravel allows you to send notifications in a locale other than the HTTP request's current locale, and will even remember this locale if the notification is queued.

To accomplish this, the `Illuminate\Notifications\Notification` class offers a `locale` method to set the desired language. The application will change into this locale when the notification is being evaluated and then revert back to the previous locale when evaluation is complete:

```
$user->notify((new InvoicePaid($invoice))->locale('es'));
```

Localization of multiple notifiable entries may also be achieved via the `Notification` facade:

```
Notification::locale('es')->send(
    $users, new InvoicePaid($invoice)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on your notifiable model, you may instruct Laravel to use this stored locale when sending a notification:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     *
     * @return string
     */
```

```
    public function preferredLocale()
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending notifications and mailables to the model. Therefore, there is no need to call the `locale` method when using this interface:

```
$user->notify(new InvoicePaid($invoice));
```

Notification Events

When a notification is sent, the `Illuminate\Notifications\Events\NotificationSent` event is fired by the notification system. This contains the "notifiable" entity and the notification instance itself. You may register listeners for this event in your `EventServiceProvider`:

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

> TIP After registering listeners in your `EventServiceProvider`, use the `event:generate` Artisan command to quickly generate listener classes.

Within an event listener, you may access the `notifiable`, `notification`, and `channel` properties on the event to learn more about the notification recipient or the notification itself:

```
/**
 * Handle the event.
 *
 * @param  \Illuminate\Notifications\Events\NotificationSent  $event
 * @return void
 */
public function handle(NotificationSent $event)
{
    // $event->channel
    // $event->notifiable
    // $event->notification
    // $event->response
}
```

Custom Channels

Laravel ships with a handful of notification channels, but you may want to write your own drivers to deliver notifications via other channels. Laravel makes it simple. To get started, define a class that contains a `send` method. The method should receive two arguments: a `$notifiable` and a `$notification`.

Within the `send` method, you may call methods on the notification to retrieve a message object understood by your channel and then send the notification to the `$notifiable` instance however you wish:

```
<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * Send the given notification.
     *
     * @param  mixed  $notifiable
     * @param  \Illuminate\Notifications\Notification  $notification
```

```
     * @return void
     */
    public function send($notifiable, Notification $notification)
    {
        $message = $notification->toVoice($notifiable);

        // Send notification to the $notifiable instance...
    }
}
```

Once your notification channel class has been defined, you may return the class name from the `via` method of any of your notifications. In this example, the `toVoice` method of your notification can return whatever object you choose to represent voice messages. For example, you might define your own `VoiceMessage` class to represent these messages:

```
<?php

namespace App\Notifications;

use App\Channels\Messages\VoiceMessage;
use App\Channels\VoiceChannel;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Get the notification channels.
     *
     * @param  mixed  $notifiable
     * @return array|string
     */
    public function via($notifiable)
    {
        return [VoiceChannel::class];
    }

    /**
     * Get the voice representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return VoiceMessage
     */
    public function toVoice($notifiable)
    {
        // ...
    }
}
```

Digging Deeper

Package Development

- Introduction
  - A Note On Facades
- Package Discovery
- Service Providers
- Resources
  - Configuration
  - Migrations
  - Routes
  - Translations
  - Views
  - View Components
- Commands
- Public Assets
- Publishing File Groups

Introduction

Packages are the primary way of adding functionality to Laravel. Packages might be anything from a great way to work with dates like Carbon or a package that allows you to associate files with Eloquent models like Spatie's Laravel Media Library.

There are different types of packages. Some packages are stand-alone, meaning they work with any PHP framework. Carbon and PHPUnit are examples of stand-alone packages. Any of these packages may be used with Laravel by requiring them in your `composer.json` file.

On the other hand, other packages are specifically intended for use with Laravel. These packages may have routes, controllers, views, and configuration specifically intended to enhance a Laravel application. This guide primarily covers the development of those packages that are Laravel specific.

A Note On Facades

When writing a Laravel application, it generally does not matter if you use contracts or facades since both provide essentially equal levels of testability. However, when writing packages, your package will not typically have access to all of Laravel's testing helpers. If you would like to be able to write your package tests as if the package were installed inside a typical Laravel application, you may use the Orchestral Testbench package.

Package Discovery

In a Laravel application's `config/app.php` configuration file, the `providers` option defines a list of service providers that should be loaded by Laravel. When someone installs your package, you will typically want your service provider to be included in this list. Instead of requiring users to manually add your service provider to the list, you may define the provider in the `extra` section of your package's `composer.json` file. In addition to service providers, you may also list any facades you would like to be registered:

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

Once your package has been configured for discovery, Laravel will automatically register its service providers and facades when it is installed, creating a convenient installation experience for your package's users.

Opting Out Of Package Discovery

If you are the consumer of a package and would like to disable package discovery for a package, you may list the package name in the `extra` section of your application's `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

You may disable package discovery for all packages using the `*` character inside of your application's `dont-discover` directive:

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

Service Providers

Service providers are the connection point between your package and Laravel. A service provider is responsible for binding things into Laravel's service container and informing Laravel where to load package resources such as views, configuration, and localization files.

A service provider extends the `Illuminate\Support\ServiceProvider` class and contains two methods: `register` and `boot`. The base `ServiceProvider` class is located in the `illuminate/support` Composer package, which you should add to your own package's dependencies. To learn more about the structure and purpose of service providers, check out their documentation.

Resources

Configuration

Typically, you will need to publish your package's configuration file to the application's `config` directory. This will allow users of your package to easily override your default configuration options. To allow your configuration files to be published, call the `publishes` method from the `boot` method of your service provider:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/courier.php' => config_path('courier.php'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` command, your file will be copied to the specified publish location. Once your configuration has been published, its values may be accessed like any other configuration file:

```
$value = config('courier.option');
```

> NOTE You should not define closures in your configuration files. They can not be serialized correctly when users execute the `config:cache` Artisan command.

Default Package Configuration

You may also merge your own package configuration file with the application's published copy. This will allow your users to define only the options they actually want to override in the published copy of the configuration file. To merge the configuration file values, use the `mergeConfigFrom` method within your service provider's `register` method.

The `mergeConfigFrom` method accepts the path to your package's configuration file as its first argument and the name of the application's copy of the configuration file as its second argument:

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```

> NOTE This method only merges the first level of the configuration array. If your users partially define a multi-dimensional configuration array, the missing options will not be merged.

Routes

If your package contains routes, you may load them using the `loadRoutesFrom` method. This method will automatically determine if the application's routes are cached and will not load your routes file if the routes have already been cached:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/../routes/web.php');
}
```

Migrations

If your package contains database migrations, you may use the `loadMigrationsFrom` method to inform Laravel how to load them. The `loadMigrationsFrom` method accepts the path to your package's migrations as its only argument:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
}
```

Once your package's migrations have been registered, they will automatically be run when the `php artisan migrate` command is executed. You do not need to export them to the application's `database/migrations` directory.

Translations

If your package contains translation files, you may use the `loadTranslationsFrom` method to inform Laravel how to load them. For example, if your package is named `courier`, you should add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 *
```

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

Package translations are referenced using the `package::file.line` syntax convention. So, you may load the `courier` package's `welcome` line from the `messages` file like so:

```
echo trans('courier::messages.welcome');
```

Publishing Translations

If you would like to publish your package's translations to the application's `resources/lang/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package paths and their desired publish locations. For example, to publish the translation files for the `courier` package, you may do the following:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/../resources/lang', 'courier');

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

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's translations will be published to the specified publish location.

Views

To register your package's views with Laravel, you need to tell Laravel where the views are located. You may do this using the service provider's `loadViewsFrom` method. The `loadViewsFrom` method accepts two arguments: the path to your view templates and your package's name. For example, if your package's name is `courier`, you would add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

Package views are referenced using the `package::view` syntax convention. So, once your view path is registered in a service provider, you may load the `admin` view from the `courier` package like so:

```
Route::get('/dashboard', function () {
    return view('courier::dashboard');
});
```

Overriding Package Views

When you use the `loadViewsFrom` method, Laravel actually registers two locations for your views: the application's `resources/views/vendor` directory and the directory you specify. So, using the `courier` package as an example, Laravel will first check if a custom version of the view has been placed in the `resources/views/vendor/courier` directory by the developer. Then, if the view has not been customized, Laravel will search the package view directory you specified in your call to `loadViewsFrom`. This makes it easy for package users to customize / override your package's views.

Publishing Views

If you would like to make your views available for publishing to the application's `resources/views/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package view paths and their desired publish locations:

```
/**
 * Bootstrap the package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

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

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's views will be copied to the specified publish location.

View Components

If your package contains view components, you may use the `loadViewComponentsAs` method to inform Laravel how to load them. The `loadViewComponentsAs` method accepts two arguments: the tag prefix for your view components and an array of your view component class names. For example, if your package's prefix is `courier` and you have `Alert` and `Button` view components, you would add the following to your service provider's `boot` method:

```
use Courier\Components\Alert;
use Courier\Components\Button;

/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewComponentsAs('courier', [
        Alert::class,
        Button::class,
    ]);
}
```

Once your view components are registered in a service provider, you may reference them in your view like so:

```
<x-courier-alert />

<x-courier-button />
```

Anonymous Components

If your package contains anonymous components, they must be placed within a `components` directory of your package's "views" directory (as specified by `loadViewsFrom`). Then, you may render them by prefixing the component name with the package's view namespace:

```
<x-courier::alert />
```

Commands

To register your package's Artisan commands with Laravel, you may use the `commands` method. This method expects an array of command class names. Once the commands have been registered, you may execute them using the Artisan CLI:

```
use Courier\Console\Commands\InstallCommand;
```

```
use Courier\Console\Commands\NetworkCommand;

/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            InstallCommand::class,
            NetworkCommand::class,
        ]);
    }
}
```

Public Assets

Your package may have assets such as JavaScript, CSS, and images. To publish these assets to the application's `public` directory, use the service provider's `publishes` method. In this example, we will also add a `public` asset group tag, which may be used to easily publish groups of related assets:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../public' => public_path('vendor/courier'),
    ], 'public');
}
```

Now, when your package's users execute the `vendor:publish` command, your assets will be copied to the specified publish location. Since users will typically need to overwrite the assets every time the package is updated, you may use the `--force` flag:

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

Publishing File Groups

You may want to publish groups of package assets and resources separately. For instance, you might want to allow your users to publish your package's configuration files without being forced to publish your package's assets. You may do this by "tagging" them when calling the `publishes` method from a package's service provider. For example, let's use tags to define two publish groups (`config` and `migrations`) in the `boot` method of a package's service provider:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'migrations');
}
```

Now your users may publish these groups separately by referencing their tag when executing the `vendor:publish` command:

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

Digging Deeper

Queues

- Introduction
  - Connections Vs. Queues
  - Driver Notes & Prerequisites
- Creating Jobs
  - Generating Job Classes
  - Class Structure
  - Unique Jobs
- Job Middleware
  - Rate Limiting
  - Preventing Job Overlaps
- Dispatching Jobs
  - Delayed Dispatching
  - Synchronous Dispatching
  - Jobs & Database Transactions
  - Job Chaining
  - Customizing The Queue & Connection
  - Specifying Max Job Attempts / Timeout Values
  - Error Handling
- Job Batching
  - Defining Batchable Jobs
  - Dispatching Batches
  - Adding Jobs To Batches
  - Inspecting Batches
  - Cancelling Batches
  - Batch Failures
  - Pruning Batches
- Queueing Closures
- Running The Queue Worker
  - The `queue:work` Command
  - Queue Priorities
  - Queue Workers & Deployment
  - Job Expirations & Timeouts
- Supervisor Configuration
- Dealing With Failed Jobs
  - Cleaning Up After Failed Jobs
  - Retrying Failed Jobs
  - Ignoring Missing Models
  - Failed Job Events
- Clearing Jobs From Queues
- Job Events

Introduction

While building your web application, you may have some tasks, such as parsing and storing an uploaded CSV file, that take too long to perform during a typical web request. Thankfully, Laravel allows you to easily create queued jobs that may be processed in the background. By moving time intensive tasks to a queue, your application can respond to web requests with blazing speed and provide a better user experience to your customers.

Laravel queues provide a unified queueing API across a variety of different queue backends, such as Amazon SQS, Redis, or even a relational database.

Laravel's queue configuration options are stored in your application's `config/queue.php` configuration file. In this file you will find connection configurations for each of the queue drivers that are included with the framework, including the database, Amazon SQS, Redis, and Beanstalkd drivers, as well as a synchronous

driver that will execute jobs immediately (for use during local development). A `null` queue driver is also included which discards queued jobs.

> TIP Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full Horizon documentation for more information.

Connections Vs. Queues

Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". In your `config/queue.php` configuration file, there is a `connections` configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.

Note that each connection configuration example in the `queue` configuration file contains a `queue` attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the `queue` attribute of the connection configuration:

```
use App\Jobs\ProcessPodcast;

// This job is sent to the default connection's default queue...
ProcessPodcast::dispatch();

// This job is sent to the default connection's "emails" queue...
ProcessPodcast::dispatch()->onQueue('emails');
```

Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a `high` queue, you may run a worker that gives them higher processing priority:

```
php artisan queue:work --queue=high,default
```

Driver Notes & Prerequisites

Database

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

```
php artisan queue:table

php artisan migrate
```

Redis

In order to use the `redis` queue driver, you should configure a Redis database connection in your `config/database.php` configuration file.

Redis Cluster

If your Redis queue connection uses a Redis Cluster, your queue names must contain a key hash tag. This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

Blocking

When using the Redis queue, you may use the `block_for` configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.

Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to `5` to indicate that the driver should block for five seconds while waiting for a job to become available:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```

NOTE Setting `block_for` to `0` will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as `SIGTERM` from being handled until the next job has been processed.

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` or phpredis PHP extension

Creating Jobs

Generating Job Classes

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. If the `app/Jobs` directory doesn't exist, it will be created when you run the `make:job` Artisan command:

```
php artisan make:job ProcessPodcast
```

The generated class will implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating to Laravel that the job should be pushed onto the queue to run asynchronously.

TIP Job stubs may be customized using stub publishing

Class Structure

Job classes are very simple, normally containing only a `handle` method that is invoked when the job is processed by the queue. To get started, let's take a look at an example job class. In this example, we'll pretend we manage a podcast publishing service and need to process the uploaded podcast files before they are published:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
```

```
     * The podcast instance.
     *
     * @var \App\Models\Podcast
     */
    protected $podcast;

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

    /**
     * Execute the job.
     *
     * @param  App\Services\AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }
}
```

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

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

`handle` Method Dependency Injection

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

If you would like to take total control over how the container injects dependencies into the `handle` method, you may use the container's `bindMethod` method. The `bindMethod` method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the `handle` method however you wish. Typically, you should call this method from the `boot` method of your `App\Providers\AppServiceProvider` service provider:

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function ($job, $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

> NOTE Binary data, such as raw image contents, should be passed through the `base64_encode` function before being passed to a queued job. Otherwise, the job may not properly serialize to JSON when being placed on the queue.

Handling Relationships

Because loaded relationships also get serialized, the serialized job string can sometimes become quite large. To prevent relations from being serialized, you can call the `withoutRelations` method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

```
/**
 * Create a new job instance.
 *
 * @param  \App\Models\Podcast  $podcast
```

```
 * @return void
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

Unique Jobs

> NOTE Unique jobs require a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the `ShouldBeUnique` interface on your job class. This interface does not require you to define any additional methods on your class:

```
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

In the example above, the `UpdateSearchIndex` job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing.

In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    /**
     * The product instance.
     *
     * @var \App\Product
     */
    public $product;

    /**
    * The number of seconds after which the job's unique lock will be released.
    *
    * @var int
    */
    public $uniqueFor = 3600;

    /**
    * The unique ID of the job.
    *
    * @return string
    */
    public function uniqueId()
    {
        return $this->product->id;
    }
}
```

In the example above, the `UpdateSearchIndex` job is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.

Keeping Jobs Unique Until Processing Begins

By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    // ...
}
```

Unique Job Locks

Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a lock with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method the returns the cache driver that should be used:

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

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...

    /**
    * Get the cache driver for the unique job lock.
    *
    * @return \Illuminate\Contracts\Cache\Repository
    */
    public function uniqueVia()
    {
        return Cache::driver('redis');
    }
}
```

> TIP If only need to limit the concurrent processing of a job, use the `WithoutOverlapping` job middleware instead.

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following `handle` method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

```
use Illuminate\Support\Facades\Redis;

/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
        info('Lock obtained...');

        // Handle job...
    }, function () {
        // Could not obtain lock...

        return $this->release(5);
    });
}
```

While this code is valid, the implementation of the `handle` method becomes noisy since it is cluttered with Redis

rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.

Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in a `app/Jobs/Middleware` directory:

```
<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * Process the queued job.
     *
     * @param  mixed  $job
     * @param  callable  $next
     * @return mixed
     */
    public function handle($job, $next)
    {
        Redis::throttle('key')
                ->block(0)->allow(1)->every(5)
                ->then(function () use ($job, $next) {
                    // Lock obtained...

                    $next($job);
                }, function () use ($job) {
                    // Could not obtain lock...

                    $job->release(5);
                });
    }
}
```

As you can see, like route middleware, job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

After creating job middleware, they may be attached to a job by returning them from the job's `middleware` method. This method does not exist on jobs scaffolded by the `make:job` Artisan command, so you will need to manually add it to your job class:

```
use App\Jobs\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited];
}
```

Rate Limiting

Although we just demonstrated how to write your own rate limiting job middleware, Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like route rate limiters, job rate limiters are defined using the `RateLimiter` facade's `for` method.

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a `RateLimiter` in the `boot` method of your `AppServiceProvider`:

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

/**
```

```
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    RateLimiter::for('backups', function ($job) {
        return $job->user->vipCustomer()
                    ? Limit::none()
                    : Limit::perHour(1)->by($job->user->id);
    });
}
```

In the example above, we defined an hourly rate limit; however, you may easily define a rate limit based on minutes using the `perMinute` method. In addition, you may pass any value you wish to the `by` method of the rate limit; however, this value is most often used to segment rate limits by customer:

```
return Limit::perMinute(50)->by($job->user->id);
```

Once you have defined your rate limit, you may attach the rate limiter to your backup job using the `Illuminate\Queue\Middleware\RateLimited` middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited('backups')];
}
```

Releasing a rate limited job back onto the queue will still increment the job's total number of `attempts`. You may wish to tune your `tries` and `maxExceptions` properties on your job class accordingly. Or, you may wish to use the `retryUntil` method to define the amount of time until the job should no longer be attempted.

> TIP If you are using Redis, you may use the `Illuminate\Queue\Middleware\RateLimitedWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

Preventing Job Overlaps

Laravel includes an `Illuminate\Queue\Middleware\WithoutOverlapping` middleware that allows you to prevent job overlaps based on an arbitrary key. This can be helpful when a queued job is modifying a resource that should only be modified by one job at a time.

For example, let's imagine you have a queued job that updates a user's credit score and you want to prevent credit score update job overlaps for the same user ID. To accomplish this, you can return the `WithoutOverlapping` middleware from your job's `middleware` method:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new WithoutOverlapping($this->user->id)];
}
```

Any overlapping jobs will be released back to the queue. You may also specify the number of seconds that must elapse before the released job will be attempted again:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
```

```
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the `dontRelease` method:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->dontRelease()];
}
```

> NOTE The `WithoutOverlapping` middleware requires a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks.

Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` method on the job itself. The arguments passed to the `dispatch` method will be given to the job's constructor:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // ...

        ProcessPodcast::dispatch($podcast);
    }
}
```

If you would like to conditionally dispatch a job, you may use the `dispatchIf` and `dispatchUnless` methods:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);

ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

Delayed Dispatching

If you would like to specify that a job should not be immediately available for processing by a queue worker, you may use the `delay` method when dispatching the job. For example, let's specify that a job should not be available for processing until 10 minutes after it has been dispatched:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
```

```
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        Podcast::create(...);

        // ...

        ProcessPodcast::dispatch($podcast)
                    ->delay(now()->addMinutes(10));
    }
}
```

NOTE The Amazon SQS queue service has a maximum delay time of 15 minutes.

Dispatching After The Response Is Sent To Browser

Alternatively, the `dispatchAfterResponse` method delays dispatching a job until after the HTTP response is sent to the user's browser. This will still allow the user to begin using the application even though a queued job is still executing. This should typically only be used for jobs that take about a second, such as sending an email. Since they are processed within the current HTTP request, jobs dispatched in this fashion do not require a queue worker to be running in order for them to be processed:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

You may also `dispatch` a closure and chain the `afterResponse` method onto the `dispatch` helper to execute a closure after the HTTP response has been sent to the browser:

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

dispatch(function () {
    Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

Synchronous Dispatching

If you would like to dispatch a job immediately (synchronously), you may use the `dispatchSync` method. When using this method, the job will not be queued and will be executed immediately within the current process:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatchSync($podcast);
```

```
    }
}
```

Jobs & Database Transactions

While it is perfectly fine to dispatch jobs within database transactions, you should take special care to ensure that your job will actually be able to execute successfully. When dispatching a job within a transaction, it is possible that the job will be processed by a worker before the transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database.

Thankfully, Laravel provides several methods of working around this problem. First, you may set the `after_commit` connection option in your queue connection's configuration array:

```
'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],
```

When the `after_commit` option is `true`, you may dispatch jobs within database transactions; however, Laravel will wait until all open database transactions have been committed before actually dispatching the job. Of course, if no database transactions are currently open, the job will be dispatched immediately.

If a transaction is rolled back due to an exception that occurs during the transaction, the dispatched jobs that were dispatched during that transaction will be discarded.

> TIP Setting the `after_commit` configuration option to `true` will also cause any queued event listeners, mailables, notifications, and broadcast events to be dispatched after all open database transactions have been committed.

Specifying Commit Dispatch Behavior Inline

If you do not set the `after_commit` queue connection configuration option to `true`, you may still indicate that a specific job should be dispatched after all open database transactions have been committed. To accomplish this, you may chain the `afterCommit` method onto your dispatch operation:

```
use App\Jobs\ProcessPodcast;

ProcessPodcast::dispatch($podcast)->afterCommit();
```

Likewise, if the `after_commit` configuration option is set to `true`, you may indicate that a specific job should be dispatched immediately without waiting for any open database transactions to commit:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

Job Chaining

Job chaining allows you to specify a list of queued jobs that should be run in sequence after the primary job has executed successfully. If one job in the sequence fails, the rest of the jobs will not be run. To execute a queued job chain, you may use the `chain` method provided by the `Bus` facade. Laravel's command bus is a lower level component that queued job dispatching is built on top of:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->dispatch();
```

In addition to chaining job class instances, you may also chain closures:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    function () {
        Podcast::update(...);
    },
])->dispatch();
```

NOTE Deleting jobs using the `$this->delete()` method within the job will not prevent chained jobs from being processed. The chain will only stop executing if a job in the chain fails.

Chain Connection & Queue

If you would like to specify the connection and queue that should be used for the chained jobs, you may use the `onConnection` and `onQueue` methods. These methods specify the queue connection and queue name that should be used unless the queued job is explicitly assigned a different connection / queue:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

Chain Failures

When chaining jobs, you may use the `catch` method to specify a closure that should be invoked if a job within the chain fails. The given callback will receive the `Throwable` instance that caused the job failure:

```
use Illuminate\Support\Facades\Bus;
use Throwable;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->catch(function (Throwable $e) {
    // A job within the chain has failed...
})->dispatch();
```

Customizing The Queue & Connection

Dispatching To A Particular Queue

By pushing jobs to different queues, you may "categorize" your queued jobs and even prioritize how many workers you assign to various queues. Keep in mind, this does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method when dispatching the job:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...
```

```
        ProcessPodcast::dispatch($podcast)->onQueue('processing');
    }
}
```

Alternatively, you may specify the job's queue by calling the `onQueue` method within the job's constructor:

```
<?php

namespace App\Jobs;

 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->onQueue('processing');
    }
}
```

Dispatching To A Particular Connection

If your application interacts with multiple queue connections, you may specify which connection to push a job to using the `onConnection` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onConnection('sqs');
    }
}
```

You may chain the `onConnection` and `onQueue` methods together to specify the connection and the queue for a job:

```
ProcessPodcast::dispatch($podcast)
              ->onConnection('sqs')
              ->onQueue('processing');
```

Alternatively, you may specify the job's connection by calling the `onConnection` method within the job's constructor:

```
<?php

namespace App\Jobs;
```

```
 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->onConnection('sqs');
    }
}
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

If one of your queued jobs is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a job may be attempted.

One approach to specifying the maximum number of times a job may be attempted is via the `--tries` switch on the Artisan command line. This will apply to all jobs processed by the worker unless the job being processed specifies a more specific number of times it may be attempted:

```
php artisan queue:work --tries=3
```

If a job exceeds its maximum number of attempts, it will be considered a "failed" job. For more information on handling failed jobs, consult the failed job documentation.

You may take a more granular approach by defining the maximum number of times a job may be attempted on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the `--tries` value provided on the command line:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

Time Based Attempts

As an alternative to defining how many times a job may be attempted before it fails, you may define a time at which the job should no longer be attempted. This allows a job to be attempted any number of times within a given time frame. To define the time at which a job should no longer be attempted, add a `retryUntil` method to your job class. This method should return a `DateTime` instance:

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addMinutes(10);
```

```
}
```

TIP You may also define a `tries` property or `retryUntil` method on your queued event listeners.

Max Exceptions

Sometimes you may wish to specify that a job may be attempted many times, but should fail if the retries are triggered by a given number of unhandled exceptions (as opposed to being released by the `release` method directly). To accomplish this, you may define a `maxExceptions` property on your job class:

```
<?php

namespace App\Jobs;

use Illuminate\Support\Facades\Redis;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 25;

    /**
     * The maximum number of unhandled exceptions to allow before failing.
     *
     * @var int
     */
    public $maxExceptions = 3;

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        Redis::throttle('key')->allow(10)->every(60)->then(function () {
            // Lock obtained, process the podcast...
        }, function () {
            // Unable to obtain lock...
            return $this->release(10);
        });
    }
}
```

In this example, the job is released for ten seconds if the application is unable to obtain a Redis lock and will continue to be retried up to 25 times. However, the job will fail if three unhandled exceptions are thrown by the job.

Timeout

NOTE The `pcntl` PHP extension must be installed in order to specify job timeouts.

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server.

The maximum number of seconds that jobs can run may be specified using the `--timeout` switch on the Artisan command line:

```
php artisan queue:work --timeout=30
```

If the job exceeds its maximum attempts by continually timing out, it will be marked as failed.

You may also define the maximum number of seconds a job should be allowed to run on the job class itself. If the timeout is specified on the job, it will take precedence over any timeout specified on the command line:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

Sometimes, IO blocking processes such as sockets or outgoing HTTP connections may not respect your specified timeout. Therefore, when using these features, you should always attempt to specify a timeout using their APIs as well. For example, when using Guzzle, you should always specify a connection and request timeout value.

Error Handling

If an exception is thrown while the job is being processed, the job will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The maximum number of attempts is defined by the `--tries` switch used on the `queue:work` Artisan command. Alternatively, the maximum number of attempts may be defined on the job class itself. More information on running the queue worker can be found below.

Manually Releasing A Job

Sometimes you may wish to manually release a job back onto the queue so that it can be attempted again at a later time. You may accomplish this by calling the `release` method:

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

    $this->release();
}
```

By default, the `release` method will release the job back onto the queue for immediate processing. However, by passing an integer to the `release` method you may instruct the queue to not make the job available for processing until a given number of seconds has elapsed:

```
$this->release(10)
```

Manually Failing A Job

Occasionally you may need to manually mark a job as "failed". To do so, you may call the `fail` method:

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

    $this->fail();
}
```

If you would like to mark your job as failed because of an exception that you have caught, you may pass the exception to the `fail` method:

```
$this->fail($exception);
```

TIP For more information on failed jobs, check out the documentation on dealing with job failures.

Job Batching

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing. Before getting started, you should create a database migration to build a table to contain meta information about your job batches, such as their completion percentage. This migration may be generated using the `queue:batches-table` Artisan command:

```
php artisan queue:batches-table

php artisan migrate
```

Defining Batchable Jobs

To define a batchable job, you should create a queueable job as normal; however, you should add the `Illuminate\Bus\Batchable` trait to the job class. This trait provides access to a `batch` method which may be used to retrieve the current batch that the job is executing within:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
    use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        if ($this->batch()->cancelled()) {
            // Determine if the batch has been cancelled...

            return;
        }

        // Import a portion of the CSV file...
    }
}
```

Dispatching Batches

To dispatch a batch of jobs, you should use the `batch` method of the `Bus` facade. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the `then`, `catch`, and `finally` methods to define completion callbacks for the batch. Each of these callbacks will receive an `Illuminate\Bus\Batch` instance when they are invoked. In this example, we will imagine we are queueing a batch of jobs that each process a given number of rows from a CSV file:

```
use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->then(function (Batch $batch) {
```

```
    // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
    // First batch job failure detected...
})->finally(function (Batch $batch) {
    // The batch has finished executing...
})->dispatch();

return $batch->id;
```

The batch's ID, which may be accessed via the `$batch->id` property, may be used to query the Laravel command bus for information about the batch after it has been dispatched.

Naming Batches

Some tools such as Laravel Horizon and Laravel Telescope may provide more user-friendly debug information for batches if batches are named. To assign an arbitrary name to a batch, you may call the `name` method while defining the batch:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import CSV')->dispatch();
```

Batch Connection & Queue

If you would like to specify the connection and queue that should be used for the batched jobs, you may use the `onConnection` and `onQueue` methods. All batched jobs must execute within the same connection and queue:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

Chains Within Batches

You may define a set of chained jobs within a batch by placing the chained jobs within an array. For example, we may execute two job chains in parallel and execute a callback when both job chains have finished processing:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

Adding Jobs To Batches

Sometimes it may be useful to add additional jobs to a batch from within a batched job. This pattern can be useful when you need to batch thousands of jobs which may take too long to dispatch during a web request. So, instead, you may wish to dispatch an initial batch of "loader" jobs that hydrate the batch with even more jobs:

```
$batch = Bus::batch([
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
])->then(function (Batch $batch) {
```

```
    // All jobs completed successfully...
})->name('Import Contacts')->dispatch();
```

In this example, we will use the `LoadImportBatch` job to hydrate the batch with additional jobs. To accomplish this, we may use the `add` method on the batch instance that may be accessed via the job's `batch` method:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```

NOTE You may only add jobs to a batch from within a job that belongs to the same batch.

Inspecting Batches

The `Illuminate\Bus\Batch` instance that is provided to batch completion callbacks has a variety of properties and methods to assist you in interacting with and inspecting a given batch of jobs:

```
// The UUID of the batch...
$batch->id;

// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;

// The number of jobs that have failed...
$batch->failedJobs;

// The number of jobs that have been processed thus far...
$batch->processedJobs();

// The completion percentage of the batch (0-100)...
$batch->progress();

// Indicates if the batch has finished executing...
$batch->finished();

// Cancel the execution of the batch...
$batch->cancel();

// Indicates if the batch has been cancelled...
$batch->cancelled();
```

Returning Batches From Routes

All `Illuminate\Bus\Batch` instances are JSON serializable, meaning you can return them directly from one of your application's routes to retrieve a JSON payload containing information about the batch, including its completion progress. This makes it convenient to display information about the batch's completion progress in your application's UI.

To retrieve a batch by its ID, you may use the `Bus` facade's `findBatch` method:

```
use Illuminate\Support\Facades\Bus;
use Illuminate\Support\Facades\Route;
```

```
Route::get('/batch/{batchId}', function (string $batchId) {
    return Bus::findBatch($batchId);
});
```

Cancelling Batches

Sometimes you may need to cancel a given batch's execution. This can be accomplished by calling the `cancel` method on the `Illuminate\Bus\Batch` instance:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

    if ($this->batch()->cancelled()) {
        return;
    }
}
```

As you may have noticed in previous examples, batched jobs should typically check to see if the batch has been cancelled at the beginning of their `handle` method:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    // Continue processing...
}
```

Batch Failures

When a batched job fails, the `catch` callback (if assigned) will be invoked. This callback is only invoked for the first job that fails within the batch.

Allowing Failures

When a job within a batch fails, Laravel will automatically mark the batch as "cancelled". If you wish, you may disable this behavior so that a job failure does not automatically mark the batch as cancelled. This may be accomplished by calling the `allowFailures` method while dispatching the batch:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->allowFailures()->dispatch();
```

Retrying Failed Batch Jobs

For convenience, Laravel provides a `queue:retry-batch` Artisan command that allows you to easily retry all of the failed jobs for a given batch. The `queue:retry-batch` command accepts the UUID of the batch whose failed jobs should be retried:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

Pruning Batches

Without pruning, the `job_batches` table can accumulate records very quickly. To mitigate this, you should schedule the `queue:prune-batches` Artisan command to run daily:

```
$schedule->command('queue:prune-batches')->daily();
```

By default, all finished batches older that are more than 24 hours old will be pruned. You may use the `hours` option when calling the command to determine how long to retain batch data. For example, the following command will delete all batches that finished over 48 hours ago:

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

Queueing Closures

Instead of dispatching a job class to the queue, you may also dispatch a closure. This is great for quick, simple tasks that need to be executed outside of the current request cycle. When dispatching closures to the queue, the closure's code content is cryptographically signed so that it can not be modified in transit:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

Using the `catch` method, you may provide a closure that should be executed if the queued closure fails to complete successfully after exhausting all of your queue's configured retry attempts:

```
use Throwable;

dispatch(function () use ($podcast) {
    $podcast->publish();
})->catch(function (Throwable $e) {
    // This job has failed...
});
```

Running The Queue Worker

The `queue:work` Command

Laravel includes an Artisan command that will start a queue worker and process new jobs as they are pushed onto the queue. You may run the worker using the `queue:work` Artisan command. Note that once the `queue:work` command has started, it will continue to run until it is manually stopped or you close your terminal:

```
php artisan queue:work
```

> TIP To keep the `queue:work` process running permanently in the background, you should use a process monitor such as Supervisor to ensure that the queue worker does not stop running.

Remember, queue workers are long-lived processes and store the booted application state in memory. As a result, they will not notice changes in your code base after they have been started. So, during your deployment process, be sure to restart your queue workers. In addition, remember that any static state created or modified by your application will not be automatically reset between jobs.

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker when you want to reload your updated code or reset the application state; however, this command is significantly less efficient than the `queue:work` command:

```
php artisan queue:listen
```

Running Multiple Queue Workers

To assign multiple workers to a queue and process jobs concurrently, you should simply start multiple `queue:work` processes. This can either be done locally via multiple tabs in your terminal or in production using your process manager's configuration settings. When using Supervisor, you may use the `numprocs` configuration value.

Specifying The Connection & Queue

You may also specify which queue connection the worker should utilize. The connection name passed to the `work` command should correspond to one of the connections defined in your `config/queue.php` configuration file:

```
php artisan queue:work redis
```

You may customize your queue worker even further by only processing particular queues for a given connection. For example, if all of your emails are processed in an `emails` queue on your `redis` queue connection, you may issue the following command to start a worker that only processes that queue:

```
php artisan queue:work redis --queue=emails
```

Processing A Specified Number Of Jobs

The `--once` option may be used to instruct the worker to only process a single job from the queue:

```
php artisan queue:work --once
```

The `--max-jobs` option may be used to instruct the worker to process the given number of jobs and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after processing a given number of jobs, releasing any memory they may have accumulated:

```
php artisan queue:work --max-jobs=1000
```

Processing All Queued Jobs & Then Exiting

The `--stop-when-empty` option may be used to instruct the worker to process all jobs and then exit gracefully. This option can be useful when processing Laravel queues within a Docker container if you wish to shutdown the container after the queue is empty:

```
php artisan queue:work --stop-when-empty
```

Processing Jobs For A Given Number Of Seconds

The `--max-time` option may be used to instruct the worker to process jobs for the given number of seconds and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after processing jobs for a given amount of time, releasing any memory they may have accumulated:

```
// Process jobs for one hour and then exit...
php artisan queue:work --max-time=3600
```

Worker Sleep Duration

When jobs are available on the queue, the worker will keep processing jobs with no delay in between them. However, the `sleep` option determines how any seconds the worker will "sleep" if there are no new jobs available. While sleeping, the worker will not process any new jobs - the jobs will be processed after the worker wakes up again.

```
php artisan queue:work --sleep=3
```

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should release any heavy resources after each job completes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done processing the image.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` configuration file you may set the default `queue` for your `redis` connection to `low`. However, occasionally you may wish to push a job to a `high` priority queue like so:

```
dispatch((new Job)->onQueue('high'));
```

To start a worker that verifies that all of the `high` queue jobs are processed before continuing to any jobs on the `low` queue, pass a comma-delimited list of queue names to the `work` command:

```
php artisan queue:work --queue=high,low
```

Queue Workers & Deployment

Since queue workers are long-lived processes, they will not notice changes to your code without being restarted. So, the simplest way to deploy an application using queue workers is to restart the workers during your deployment process. You may gracefully restart all of the workers by issuing the `queue:restart` command:

```
php artisan queue:restart
```

This command will instruct all queue workers to gracefully exit after they finish processing their current job so that no existing jobs are lost. Since the queue workers will exit when the `queue:restart` command is executed, you should be running a process manager such as Supervisor to automatically restart the queue workers.

> TIP The queue uses the cache to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Job Expirations & Timeouts

Job Expiration

In your `config/queue.php` configuration file, each queue connection defines a `retry_after` option. This option specifies how many seconds the queue connection should wait before retrying a job that is being processed. For example, if the value of `retry_after` is set to `90`, the job will be released back onto the queue if it has been processing for 90 seconds without being released or deleted. Typically, you should set the `retry_after` value to the maximum number of seconds your jobs should reasonably take to complete processing.

> NOTE The only queue connection which does not contain a `retry_after` value is Amazon SQS. SQS will retry the job based on the Default Visibility Timeout which is managed within the AWS console.

Worker Timeouts

The `queue:work` Artisan command exposes a `--timeout` option. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server:

```
php artisan queue:work --timeout=60
```

The `retry_after` configuration option and the `--timeout` CLI option are different, but work together to ensure that jobs are not lost and that jobs are only successfully processed once.

> NOTE The `--timeout` value should always be at least several seconds shorter than your `retry_after` configuration value. This will ensure that a worker processing a frozen job is always terminated before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

Supervisor Configuration

In production, you need a way to keep your `queue:work` processes running. A `queue:work` process may stop running for a variety of reasons, such as an exceeded worker timeout or the execution of the `queue:restart` command.

For this reason, you need to configure a process monitor that can detect when your `queue:work` processes exit and automatically restart them. In addition, process monitors can allow you to specify how many `queue:work` processes you would like to run concurrently. Supervisor is a process monitor commonly used in Linux environments and we will discuss how to configure it in the following documentation.

Installing Supervisor

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

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

> TIP If configuring and managing Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your production Laravel projects.

Configuring Supervisor

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

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

In this example, the `numprocs` directive will instruct Supervisor to run eight `queue:work` processes and monitor all of them, automatically restarting them if they fail. You should change the `command` directive of the configuration to reflect your desired queue connection and worker options.

> NOTE You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker:*
```

For more information on Supervisor, consult the Supervisor documentation.

Dealing With Failed Jobs

Sometimes your queued jobs will fail. Don't worry, things don't always go as planned! Laravel includes a convenient way to specify the maximum number of times a job should be attempted. After a job has exceeded this number of attempts, it will be inserted into the `failed_jobs` database table. Of course, we will need to create that table if it does not already exist. To create a migration for the `failed_jobs` table, you may use the `queue:failed-table` command:

```
php artisan queue:failed-table

php artisan migrate
```

When running a queue worker process, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:work` command. If you do not specify a value for the `--tries` option, jobs will only be attempted once or as many times as specified by the job class' `$tries` property:

```
php artisan queue:work redis --tries=3
```

Using the `--backoff` option, you may specify how many seconds Laravel should wait before retrying a job that has encountered an exception. By default, a job is immediately released back onto the queue so that it may be attempted again:

```
php artisan queue:work redis --tries=3 --backoff=3
```

If you would like to configure how many seconds Laravel should wait before retrying a job that has encountered an exception on a per-job basis, you may do so by defining a `backoff` property on your job class:

```
/**
 * The number of seconds to wait before retrying the job.
 *
 * @var int
 */
public $backoff = 3;
```

If you require more complex logic for determining the job's backoff time, you may define a `backoff` method on your job class:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return int
*/
public function backoff()
{
    return 3;
}
```

You may easily configure "exponential" backoffs by returning an array of backoff values from the `backoff` method. In this example, the retry delay will be 1 second for the first retry, 5 seconds for the second retry, and 10 seconds for the third retry:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return array
*/
public function backoff()
{
    return [1, 5, 10];
}
```

Cleaning Up After Failed Jobs

When a particular job fails, you may want to send an alert to your users or revert any actions that were partially completed by the job. To accomplish this, you may define a `failed` method on your job class. The `Throwable` instance that caused the job to fail will be passed to the `failed` method:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The podcast instance.
     *
     * @var \App\Podcast
     */
    protected $podcast;

    /**
     * Create a new job instance.
```

```
     *
     * @param  \App\Models\Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  \App\Services\AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }

    /**
     * Handle a job failure.
     *
     * @param  \Throwable  $exception
     * @return void
     */
    public function failed(Throwable $exception)
    {
        // Send user notification of failure, etc...
    }
}
```

Retrying Failed Jobs

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

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, failure time, and other information about the job. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of `5`, issue the following command:

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

If necessary, you may pass multiple IDs or an ID range (when using numeric IDs) to the command:

```
php artisan queue:retry 5 6 7 8 9 10
```

```
php artisan queue:retry --range=5-10
```

To retry all of your failed jobs, execute the `queue:retry` command and pass `all` as the ID:

```
php artisan queue:retry all
```

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

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

> TIP When using Horizon, you should use the `horizon:forget` command to delete a failed job instead of the `queue:forget` command.

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

```
php artisan queue:flush
```

Ignoring Missing Models

When injecting an Eloquent model into a job, the model is automatically serialized before being placed on the queue and re-retrieved from the database when the job is processed. However, if the model has been deleted while the job was waiting to be processed by a worker, your job may fail with a `ModelNotFoundException`.

For convenience, you may choose to automatically delete jobs with missing models by setting your job's `deleteWhenMissingModels` property to `true`. When this model is set to `true`, Laravel will quietly discard the job without raising an exception:

```
/**
 * Delete the job if its models no longer exist.
 *
 * @var bool
 */
public $deleteWhenMissingModels = true;
```

Failed Job Events

If you would like to register an event listener that will be invoked when a job fails, you may use the `Queue` facade's `failing` method. For example, we may attach a closure to this event from the `boot` method of the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
}
```

Clearing Jobs From Queues

> TIP When using Horizon, you should use the `horizon:clear` command to clear jobs from the queue instead of the `queue:clear` command.

If you would like to delete all jobs from the default queue of the default connection, you may do so using the `queue:clear` Artisan command:

```
php artisan queue:clear
```

You may also provide the `connection` argument and `queue` option to delete jobs from a specific connection and queue:

```
php artisan queue:clear redis --queue=emails
```

> NOTE Clearing jobs from queues is only available for the SQS, Redis, and database queue drivers. In addition, the SQS message deletion process takes up to 60 seconds, so jobs sent to the SQS queue up to 60 seconds after you clear the queue might also be deleted.

Job Events

Using the `before` and `after` methods on the `Queue` facade, you may specify callbacks to be executed before or after a queued job is processed. These callbacks are a great opportunity to perform additional logging or increment statistics for a dashboard. Typically, you should call these methods from the `boot` method of a service provider. For example, we may use the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }
}
```

Using the `looping` method on the `Queue` facade, you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a closure to rollback any transactions that were left open by a previously failed job:

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

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

Digging Deeper

Task Scheduling

- Introduction
- Defining Schedules
  - Scheduling Artisan Commands
  - Scheduling Queued Jobs
  - Scheduling Shell Commands
  - Schedule Frequency Options
  - Timezones
  - Preventing Task Overlaps
  - Running Tasks On One Server
  - Background Tasks
  - Maintenance Mode
- Running The Scheduler
  - Running The Scheduler Locally
- Task Output
- Task Hooks

Introduction

In the past, you may have written a cron configuration entry for each task you needed to schedule on your server. However, this can quickly become a pain because your task schedule is no longer in source control and you must SSH into your server to view your existing cron entries or add additional entries.

Laravel's command scheduler offers a fresh approach to managing scheduled tasks on your server. The scheduler allows you to fluently and expressively define your command schedule within your Laravel application itself. When using the scheduler, only a single cron entry is needed on your server. Your task schedule is defined in the `app/Console/Kernel.php` file's `schedule` method. To help you get started, a simple example is defined within the method.

Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of your application's `App\Console\Kernel` class. To get started, let's take a look at an example. In this example, we will schedule a closure to be called every day at midnight. Within the closure we will execute a database query to clear a table:

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        //
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
```

```
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

In addition to scheduling using closures, you may also schedule invokable objects. Invokable objects are simple PHP classes that contain an `__invoke` method:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

If you would like to view an overview of your scheduled tasks and the next time they are scheduled to run, you may use the `schedule:list` Artisan command:

```
php artisan schedule:list
```

Scheduling Artisan Commands

In addition to scheduling closures, you may also schedule Artisan commands and system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class.

When scheduling Artisan commands using the command's class name, you may pass an array of additional command-line arguments that should be provided to the command when it is invoked:

```
use App\Console\Commands\SendEmailCommand;

$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(EmailsCommand::class, ['Taylor', '--force'])->daily();
```

Scheduling Queued Jobs

The `job` method may be used to schedule a queued job. This method provides a convenient way to schedule queued jobs without using the `call` method to define closures to queue the job:

```
use App\Jobs\Heartbeat;

$schedule->job(new Heartbeat)->everyFiveMinutes();
```

Optional second and third arguments may be provided to the `job` method which specify the queue name and queue connection that should be used to queue the job:

```
use App\Jobs\Heartbeat;

// Dispatch the job to the "heartbeats" queue on the "sqs" connection...
$schedule->job(new Heartbeat, 'heartbeats', 'sqs')->everyFiveMinutes();
```

Scheduling Shell Commands

The `exec` method may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

We've already seen a few examples of how you may configure a task to run at specified intervals. However, there are many more task schedule frequencies that you may assign to a task:

Method	Description
`->cron('* * * * *');`	Run the task on a custom cron schedule
`->everyMinute();`	Run the task every minute
`->everyTwoMinutes();`	Run the task every two minutes
`->everyThreeMinutes();`	Run the task every three minutes
`->everyFourMinutes();`	Run the task every four minutes

`->everyFiveMinutes();`	Run the task every five minutes
`->everyTenMinutes();`	Run the task every ten minutes
`->everyFifteenMinutes();`	Run the task every fifteen minutes
`->everyThirtyMinutes();`	Run the task every thirty minutes
`->hourly();`	Run the task every hour
`->hourlyAt(17);`	Run the task every hour at 17 minutes past the hour
`->everyTwoHours();`	Run the task every two hours
`->everyThreeHours();`	Run the task every three hours
`->everyFourHours();`	Run the task every four hours
`->everySixHours();`	Run the task every six hours
`->daily();`	Run the task every day at midnight
`->dailyAt('13:00');`	Run the task every day at 13:00
`->twiceDaily(1, 13);`	Run the task daily at 1:00 & 13:00
`->weekly();`	Run the task every Sunday at 00:00
`->weeklyOn(1, '8:00');`	Run the task every week on Monday at 8:00
`->monthly();`	Run the task on the first day of every month at 00:00
`->monthlyOn(4, '15:00');`	Run the task every month on the 4th at 15:00
`->twiceMonthly(1, 16, '13:00');`	Run the task monthly on the 1st and 16th at 13:00
`->lastDayOfMonth('15:00');`	Run the task on the last day of the month at 15:00
`->quarterly();`	Run the task on the first day of every quarter at 00:00
`->yearly();`	Run the task on the first day of every year at 00:00
`->yearlyOn(6, 1, '17:00');`	Run the task every year on June 1st at 17:00
`->timezone('America/New_York');`	Set the timezone for the task

These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, you may schedule a command to run weekly on Monday:

```
// Run once per week on Monday at 1 PM...
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')
          ->weekdays()
          ->hourly()
          ->timezone('America/Chicago')
          ->between('8:00', '17:00');
```

A list of additional schedule constraints may be found below:

Method	Description
`->weekdays();`	Limit the task to weekdays
`->weekends();`	Limit the task to weekends
`->sundays();`	Limit the task to Sunday
`->mondays();`	Limit the task to Monday
`->tuesdays();`	Limit the task to Tuesday
`->wednesdays();`	Limit the task to Wednesday
`->thursdays();`	Limit the task to Thursday
`->fridays();`	Limit the task to Friday
`->saturdays();`	Limit the task to Saturday
`->days(array|mixed);`	Limit the task to specific days
`->between($startTime, $endTime);`	Limit the task to run between start and end times
`->unlessBetween($startTime, $endTime);`	Limit the task to not run between start and end times
`->when(Closure);`	Limit the task based on a truth test
`->environments($env);`	Limit the task to specific environments

Day Constraints

The `days` method may be used to limit the execution of a task to specific days of the week. For example, you may schedule a command to run hourly on Sundays and Wednesdays:

```
$schedule->command('emails:send')
                ->hourly()
                ->days([0, 3]);
```

Alternatively, you may use the constants available on the `Illuminate\Console\Scheduling\Schedule` class when defining the days on which a task should run:

```
use Illuminate\Console\Scheduling\Schedule;

$schedule->command('emails:send')
                ->hourly()
                ->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

Between Time Constraints

The `between` method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('emails:send')
                    ->hourly()
                    ->between('7:00', '22:00');
```

Similarly, the `unlessBetween` method can be used to exclude the execution of a task for a period of time:

```
$schedule->command('emails:send')
                    ->hourly()
                    ->unlessBetween('23:00', '4:00');
```

Truth Test Constraints

The `when` method may be used to limit the execution of a task based on the result of a given truth test. In other words, if the given closure returns `true`, the task will execute as long as no other constraining conditions prevent the task from running:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

The `skip` method may be seen as the inverse of `when`. If the `skip` method returns `true`, the scheduled task will not be executed:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return `true`.

Environment Constraints

The `environments` method may be used to execute tasks only on the given environments (as defined by the `APP_ENV` environment variable):

```
$schedule->command('emails:send')
            ->daily()
            ->environments(['staging', 'production']);
```

Timezones

Using the `timezone` method, you may specify that a scheduled task's time should be interpreted within a given timezone:

```
$schedule->command('report:generate')
         ->timezone('America/New_York')
         ->at('2:00')
```

If you are repeatedly assigning the same timezone to all of your scheduled tasks, you may wish to define a

`scheduleTimezone` method in your `App\Console\Kernel` class. This method should return the default timezone that should be assigned to all scheduled tasks:

```
/**
 * Get the timezone that should be used by default for scheduled events.
 *
 * @return \DateTimeZone|string|null
 */
protected function scheduleTimezone()
{
    return 'America/Chicago';
}
```

> NOTE Remember that some timezones utilize daylight savings time. When daylight saving time changes occur, your scheduled task may run twice or even not run at all. For this reason, we recommend avoiding timezone scheduling when possible.

Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` Artisan command will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

If needed, you may specify how many minutes must pass before the "without overlapping" lock expires. By default, the lock will expire after 24 hours:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

Running Tasks On One Server

> NOTE To utilize this feature, your application must be using the `database`, `memcached`, `dynamodb`, or `redis` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

If your application's scheduler is running on multiple servers, you may limit a scheduled job to only execute on a single server. For instance, assume you have a scheduled task that generates a new report every Friday night. If the task scheduler is running on three worker servers, the scheduled task will run on all three servers and generate the report three times. Not good!

To indicate that the task should run on only one server, use the `onOneServer` method when defining the scheduled task. The first server to obtain the task will secure an atomic lock on the job to prevent other servers from running the same task at the same time:

```
$schedule->command('report:generate')
                ->fridays()
                ->at('17:00')
                ->onOneServer();
```

Background Tasks

By default, multiple tasks scheduled at the same time will execute sequentially based on the order they are defined in your `schedule` method. If you have long-running tasks, this may cause subsequent tasks to start much later than anticipated. If you would like to run tasks in the background so that they may all run simultaneously, you may use the `runInBackground` method:

```
$schedule->command('analytics:report')
         ->daily()
         ->runInBackground();
```

> NOTE The `runInBackground` method may only be used when scheduling tasks via the `command` and `exec`

methods.

Maintenance Mode

Your application's scheduled tasks will not run when the application is in maintenance mode, since we don't want your tasks to interfere with any unfinished maintenance you may be performing on your server. However, if you would like to force a task to run even in maintenance mode, you may call the `evenInMaintenanceMode` method when defining the task:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

Running The Scheduler

Now that we have learned how to define scheduled tasks, let's discuss how to actually run them on our server. The `schedule:run` Artisan command will evaluate all of your scheduled tasks and determine if they need to run based on the server's current time.

So, when using Laravel's scheduler, we only need to add a single cron configuration entry to our server that runs the `schedule:run` command every minute. If you do not know how to add cron entries to your server, consider using a service such as Laravel Forge which can manage the cron entries for you:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

Running The Scheduler Locally

Typically, you would not add a scheduler cron entry to your local development machine. Instead you may use the `schedule:work` Artisan command. This command will run in the foreground and invoke the scheduler every minute until you terminate the command:

```
php artisan schedule:work
```

Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
         ->daily()
         ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
         ->daily()
         ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may email the output to an email address of your choice. Before emailing the output of a task, you should configure Laravel's email services:

```
$schedule->command('report:generate')
         ->daily()
         ->sendOutputTo($filePath)
         ->emailOutputTo('taylor@example.com');
```

If you only want to email the output if the scheduled Artisan or system command terminates with a non-zero exit code, use the `emailOutputOnFailure` method:

```
$schedule->command('report:generate')
         ->daily()
         ->emailOutputOnFailure('taylor@example.com');
```

NOTE The `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo`, and `appendOutputTo` methods are exclusive to the `command` and `exec` methods.

Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is executed:

```
$schedule->command('emails:send')
         ->daily()
         ->before(function () {
             // The task is about to execute...
         })
         ->after(function () {
             // The task has executed...
         });
```

The `onSuccess` and `onFailure` methods allow you to specify code to be executed if the scheduled task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
         ->daily()
         ->onSuccess(function () {
             // The task succeeded...
         })
         ->onFailure(function () {
             // The task failed...
         });
```

If output is available from your command, you may access it in your `after`, `onSuccess` or `onFailure` hooks by type-hinting an `Illuminate\Support\Stringable` instance as the `$output` argument of your hook's closure definition:

```
use Illuminate\Support\Stringable;

$schedule->command('emails:send')
         ->daily()
         ->onSuccess(function (Stringable $output) {
             // The task succeeded...
         })
         ->onFailure(function (Stringable $output) {
             // The task failed...
         });
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is executed. This method is useful for notifying an external service, such as Envoyer, that your scheduled task is beginning or has finished execution:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBefore($url)
         ->thenPing($url);
```

The `pingBeforeIf` and `thenPingIf` methods may be used to ping a given URL only if a given condition is `true`:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBeforeIf($condition, $url)
         ->thenPingIf($condition, $url);
```

The `pingOnSuccess` and `pingOnFailure` methods may be used to ping a given URL only if the task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
         ->daily()
         ->pingOnSuccess($successUrl)
         ->pingOnFailure($failureUrl);
```

All of the ping methods require the Guzzle HTTP library. Guzzle is typically installed in all new Laravel projects by default, but, you may manually install Guzzle into your project using the Composer package manager if it has been accidentally removed:

```
composer require guzzlehttp/guzzle
```

Security

Authentication

- Introduction
  - Starter Kits
  - Database Considerations
  - Ecosystem Overview
- Authentication Quickstart
  - Install A Starter Kit
  - Retrieving The Authenticated User
  - Protecting Routes
  - Login Throttling
- Manually Authenticating Users
  - Remembering Users
  - Other Authentication Methods
- HTTP Basic Authentication
  - Stateless HTTP Basic Authentication
- Logging Out
  - Invalidating Sessions On Other Devices
- Password Confirmation
  - Configuration
  - Routing
  - Protecting Routes
- Adding Custom Guards
  - Closure Request Guards
- Adding Custom User Providers
  - The User Provider Contract
  - The Authenticatable Contract
- Social Authentication
- Events

Introduction

Many web applications provide a way for their users to authenticate with the application and "login". Implementing this feature in web applications can be a complex and potentially risky endeavor. For this reason, Laravel strives to give you the tools you need to implement authentication quickly, securely, and easily.

At its core, Laravel's authentication facilities are made up of "guards" and "providers". Guards define how users are authenticated for each request. For example, Laravel ships with a `session` guard which maintains state using session storage and cookies.

Providers define how users are retrieved from your persistent storage. Laravel ships with support for retrieving users using Eloquent and the database query builder. However, you are free to define additional providers as needed for your application.

Your application's authentication configuration file is located at `config/auth.php`. This file contains several well documented options for tweaking the behavior of Laravel's authentication services.

Starter Kits

Want to get started fast? Install a Laravel application starter kit in a fresh Laravel application. After migrating your database, navigate your browser to `/register` or any other URL that is assigned to your application. The starter kits will take care of scaffolding your entire authentication system!

Even if you choose to not use a starter kit in your final Laravel application, installing the Laravel Breeze starter kit can be a wonderful opportunity to learn how to implement all of Laravel's authentication functionality in an actual Laravel project. Since Laravel Breeze creates authentication controllers, routes, and views for you, you can examine the code within these files to learn how Laravel's authentication features

may be implemented.

Database Considerations

By default, Laravel includes an `App\Models\User` Eloquent model in your `app/Models` directory. This model may be used with the default Eloquent authentication driver. If your application is not using Eloquent, you may use the `database` authentication provider which uses the Laravel query builder.

When building the database schema for the `App\Models\User` model, make sure the password column is at least 60 characters in length. Of course, the `users` table migration that is included in new Laravel applications already creates a column that exceeds this length.

Also, you should verify that your `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for users that select the "remember me" option when logging into your application. Again, the default `users` table migration that is included in new Laravel applications already contains this column.

Ecosystem Overview

Laravel offers several packages related to authentication. Before continuing, we'll review the general authentication ecosystem in Laravel and discuss each package's intended purpose.

First, consider how authentication works. When using a web browser, a user will provide their username and password via a login form. If these credentials are correct, the application will store information about the authenticated user in the user's session. A cookie issued to the browser contains the session ID so that subsequent requests to the application can associate the user with the correct session. After the session cookie is received, the application will retrieve the session data based on the session ID, note that the authentication information has been stored in the session, and will consider the user as "authenticated".

When a remote service needs to authenticate to access an API, cookies are not typically used for authentication because there is no web browser. Instead, the remote service sends an API token to the API on each request. The application may validate the incoming token against a table of valid API tokens and "authenticate" the request as being performed by the user associated with that API token.

Laravel's Built-in Browser Authentication Services

Laravel includes built-in authentication and session services which are typically accessed via the `Auth` and `Session` facades. These features provide cookie based authentication for requests that are initiated from web browsers. They provide methods that allow you to verify a user's credentials and authenticate the user. In addition, these services will automatically store the proper authentication data in the user's session and issue the user's session cookie. A discussion of how to use these services is contained within this documentation.

Application Starter Kits

As discussed in this documentation, you can interact with these authentication services manually to build your application's own authentication layer. However, to help you get started more quickly, we have released free packages that provide robust, modern scaffolding of the entire authentication layer. These packages are Laravel Breeze, Laravel Jetstream, and Laravel Fortify.

Laravel Breeze is a simple, minimal implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is comprised of simple Blade templates styled with Tailwind CSS. To get started, check out the documentation on Laravel's application starter kits.

Laravel Fortify is a headless authentication backend for Laravel that implements many of the features found in this documentation, including cookie-based authentication as well as other features such as two-factor authentication and email verification. Fortify provides the authentication backend for Laravel Jetstream or may be used independently in combination with Laravel Sanctum to provide authentication for an SPA that needs to authenticate with Laravel.

Laravel Jetstream is a robust application starter kit that consumes and exposes Laravel Fortify's authentication

services with a beautiful, modern UI powered by Tailwind CSS, Livewire, and / or Inertia.js. Laravel Jetstream includes optional support for two-factor authentication, team support, browser session management, profile management, and built-in integration with Laravel Sanctum to offer API token authentication. Laravel's API authentication offerings are discussed below.

Laravel's API Authentication Services

Laravel provides two optional packages to assist you in managing API tokens and authenticating requests made with API tokens: Passport and Sanctum. Please note that these libraries and Laravel's built-in cookie based authentication libraries are not mutually exclusive. These libraries primarily focus on API token authentication while the built-in authentication services focus on cookie based browser authentication. Many applications will use both Laravel's built-in cookie based authentication services and one of Laravel's API authentication packages.

Passport

Passport is an OAuth2 authentication provider, offering a variety of OAuth2 "grant types" which allow you to issue various types of tokens. In general, this is a robust and complex package for API authentication. However, most applications do not require the complex features offered by the OAuth2 spec, which can be confusing for both users and developers. In addition, developers have been historically confused about how to authenticate SPA applications or mobile applications using OAuth2 authentication providers like Passport.

Sanctum

In response to the complexity of OAuth2 and developer confusion, we set out to build a simpler, more streamlined authentication package that could handle both first-party web requests from a web browser and API requests via tokens. This goal was realized with the release of Laravel Sanctum, which should be considered the preferred and recommended authentication package for applications that will be offering a first-party web UI in addition to an API, or will be powered by a single-page application (SPA) that exists separately from the backend Laravel application, or applications that offer a mobile client.

Laravel Sanctum is a hybrid web / API authentication package that can manage your application's entire authentication process. This is possible because when Sanctum based applications receive a request, Sanctum will first determine if the request includes a session cookie that references an authenticated session. Sanctum accomplishes this by calling Laravel's built-in authentication services which we discussed earlier. If the request is not being authenticated via a session cookie, Sanctum will inspect the request for an API token. If an API token is present, Sanctum will authenticate the request using that token. To learn more about this process, please consult Sanctum's "how it works" documentation.

Laravel Sanctum is the API package we have chosen to include with the Laravel Jetstream application starter kit because we believe it is the best fit for the majority of web application's authentication needs.

Summary & Choosing Your Stack

In summary, if your application will be accessed using a browser and you are building a monolithic Laravel application, your application will use Laravel's built-in authentication services.

Next, if your application offers an API that will be consumed by third parties, you will choose between Passport or Sanctum to provide API token authentication for your application. In general, Sanctum should be preferred when possible since it is a simple, complete solution for API authentication, SPA authentication, and mobile authentication, including support for "scopes" or "abilities".

If you are building a single-page application (SPA) that will be powered by a Laravel backend. You should use Laravel Sanctum. When using Sanctum, you will either need to manually implement your own backend authentication routes or utilize Laravel Fortify as a headless authentication backend service that provides routes and controllers for features such as registration, password reset, email verification, and more.

Passport may be chosen when your application absolutely needs all of the features provided by the OAuth2 specification.

And, if you would like to get started quickly, we are pleased to recommend Laravel Jetstream as a quick way to

start a new Laravel application that already uses our preferred authentication stack of Laravel's built-in authentication services and Laravel Sanctum.

Authentication Quickstart

> NOTE This portion of the documentation discusses authenticating users via the Laravel application starter kits, which includes UI scaffolding to help you get started quickly. If you would like to integrate with Laravel's authentication systems directly, check out the documentation on manually authenticating users.

Install A Starter Kit

First, you should install a Laravel application starter kit. Our current starter kits, Laravel Breeze and Laravel Jetstream, offer beautifully designed starting points for incorporating authentication into your fresh Laravel application.

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is made up of simple Blade templates styled with Tailwind CSS.

Laravel Jetstream is a more robust application starter kit that includes support for scaffolding your application with Livewire or Inertia.js and Vue. In addition, Jetstream features optional support for two-factor authentication, teams, profile management, browser session management, API support via Laravel Sanctum, account deletion, and more.

Retrieving The Authenticated User

After installing an authentication starter kit and allowing users to register and authenticate with your application, you will often need to interact with the currently authenticated user. While handling an incoming request, you may access the authenticated user via the `Auth` facade's `user` method:

```
use Illuminate\Support\Facades\Auth;

// Retrieve the currently authenticated user...
$user = Auth::user();

// Retrieve the currently authenticated user's ID...
$id = Auth::id();
```

Alternatively, once a user is authenticated, you may access the authenticated user via an `Illuminate\Http\Request` instance. Remember, type-hinted classes will automatically be injected into your controller methods. By type-hinting the `Illuminate\Http\Request` object, you may gain convenient access to the authenticated user from any controller method in your application via the request's `user` method:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Update the flight information for an existing flight.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        // $request->user()
    }
}
```

Determining If The Current User Is Authenticated

To determine if the user making the incoming HTTP request is authenticated, you may use the `check` method on

the `Auth` facade. This method will return `true` if the user is authenticated:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

> TIP Even though it is possible to determine if a user is authenticated using the `check` method, you will typically use a middleware to verify that the user is authenticated before allowing the user access to certain routes / controllers. To learn more about this, check out the documentation on protecting routes.

Protecting Routes

Route middleware can be used to only allow authenticated users to access a given route. Laravel ships with an `auth` middleware, which references the `Illuminate\Auth\Middleware\Authenticate` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth');
```

Redirecting Unauthenticated Users

When the `auth` middleware detects an unauthenticated user, it will redirect the user to the `login` named route. You may modify this behavior by updating the `redirectTo` function in your application's `app/Http/Middleware/Authenticate.php` file:

```
/**
 * Get the path the user should be redirected to.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return string
 */
protected function redirectTo($request)
{
    return route('login');
}
```

Specifying A Guard

When attaching the `auth` middleware to a route, you may also specify which "guard" should be used to authenticate the user. The guard specified should correspond to one of the keys in the `guards` array of your `auth.php` configuration file:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth:admin');
```

Login Throttling

If you are using the Laravel Breeze or Laravel Jetstream starter kits, rate limiting will automatically be applied to login attempts. By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts. The throttling is unique to the user's username / email address and their IP address.

> TIP If you would like to rate limit other routes in your application, check out the rate limiting documentation.

Manually Authenticating Users

You are not required to use the authentication scaffolding included with Laravel's application starter kits. If you choose to not use this scaffolding, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the `Auth` facade, so we'll need to make sure to import the `Auth` facade at the top of the class. Next, let's check out the `attempt` method. The `attempt` method is normally used to handle authentication attempt's from your application's "login" form. If authentication is successful, you should regenerate the user's session to prevent session fixation:

```
<?php

namespace App\Http\Controllers;

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

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @param  \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function authenticate(Request $request)
    {
        $credentials = $request->only('email', 'password');

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ]);
    }
}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the `password` value passed to the method via the array. You should not hash the incoming request's `password` value, since the framework will automatically hash the value before comparing it to the hashed password in the database. If the two hashed passwords match an authenticated session will be started for the user.

Remember, Laravel's authentication services will retrieve users from your database based on your authentication guard's "provider" configuration. In the default `config/auth.php` configuration file, the Eloquent user provider is specified and it is instructed to use the `App\Models\User` model when retrieving users. You may change these values within your configuration file based on the needs of your application.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

The `intended` method provided by Laravel's redirector will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware. A fallback URI may be given to this method in case the intended destination is not available.

Specifying Additional Conditions

If you wish, you may also add extra query conditions to the authentication query in addition to the user's email and password. To accomplish this, we may simply add the query conditions to the array passed to the `attempt` method. For example, we may verify that the user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // Authentication was successful...
}
```

> NOTE In these examples, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database table.

Accessing Specific Guard Instances

Via the `Auth` facade's `guard` method, you may specify which guard instance you would like to utilize when authenticating the user. This allows you to manage authentication for separate parts of your application using entirely separate authenticatable models or user tables.

The guard name passed to the `guard` method should correspond to one of the guards configured in your `auth.php` configuration file:

```
if (Auth::guard('admin')->attempt($credentials)) {
    // ...
}
```

Remembering Users

Many web applications provide a "remember me" checkbox on their login form. If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method.

When this value is `true`, Laravel will keep the user authenticated indefinitely or until they manually logout. Your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token. The `users` table migration included with new Laravel applications already includes this column:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

Other Authentication Methods

Authenticate A User Instance

If you need to set an existing user instance as the currently authenticated user, you may pass the user instance to the `Auth` facade's `login` method. The given user instance must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` contract. The `App\Models\User` model included with Laravel already implements this interface. This method of authentication is useful when you already have a valid user instance, such as directly after a user registers with your application:

```
use Illuminate\Support\Facades\Auth;

Auth::login($user);
```

You may pass a boolean value as the second argument to the `login` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::login($user, $remember = true);
```

If needed, you may specify an authentication guard before calling the `login` method:

```
Auth::guard('admin')->login($user);
```

Authenticate A User By ID

To authenticate a user using their database record's primary key, you may use the `loginUsingId` method. This method accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);
```

You may pass a boolean value as the second argument to the `loginUsingId` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::loginUsingId(1, $remember = true);
```

Authenticate A User Once

You may use the `once` method to authenticate a user with the application for a single request. No sessions or cookies will be utilized when calling this method:

```
if (Auth::once($credentials)) {
    //
}
```

HTTP Basic Authentication

HTTP Basic Authentication provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the `auth.basic` middleware to a route. The `auth.basic` middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('/profile', function () {
    // Only authenticated users may access this route...
})->middleware('auth.basic');
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the `auth.basic` middleware will assume the `email` column on your `users` database table is the user's "username".

A Note On FastCGI

If you are using PHP FastCGI and Apache to serve your Laravel application, HTTP Basic authentication may not work correctly. To correct these problems, the following lines may be added to your application's `.htaccess` file:

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session. This is primarily helpful if you choose to use HTTP Authentication to authenticate requests to your application's API. To accomplish this, define a middleware that calls the `onceBasic` method. If no response is returned by the `onceBasic` method, the request may be passed further into the application:

```
<?php

namespace App\Http\Middleware;

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }

}
```

Next, register the route middleware and attach it to a route:

```
Route::get('/api/user', function () {
    // Only authenticated users may access this route...
})->middleware('auth.basic.once');
```

Logging Out

To manually log users out of your application, you may use the `logout` method provided by the `Auth` facade.

This will remove the authentication information from the user's session so that subsequent requests are not authenticated.

In addition to calling the `logout` method, it is recommended that you invalidate the user's session and regenerate their CSRF token. After logging the user out, you would typically redirect the user to the root of your application:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Log the user out of the application.
 *
 * @param  \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function logout(Request $request)
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

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

Invalidating Sessions On Other Devices

Laravel also provides a mechanism for invalidating and "logging out" a user's sessions that are active on other devices without invalidating the session on their current device. This feature is typically utilized when a user is changing or updating their password and you would like to invalidate sessions on other devices while keeping the current device authenticated.

Before getting started, you should make sure that the `Illuminate\Session\Middleware\AuthenticateSession` middleware is present and un-commented in your `App\Http\Kernel` class' `web` middleware group:

```
'web' => [
    // ...
    \Illuminate\Session\Middleware\AuthenticateSession::class,
    // ...
],
```

Then, you may use the `logoutOtherDevices` method provided by the `Auth` facade. This method requires the user to confirm their current password, which your application should accept through an input form:

```
use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($currentPassword);
```

When the `logoutOtherDevices` method is invoked, the user's other sessions will be invalidated entirely, meaning they will be "logged out" of all guards they were previously authenticated by.

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed or before the user is redirected to a sensitive area of the application. Laravel includes built-in middleware to make this process a breeze. Implementing this feature will require you to define two routes: one route to display a view asking the user to confirm their password and another route to confirm that the password is valid and redirect the user to their intended destination.

> TIP The following documentation discusses how to integrate with Laravel's password confirmation features directly; however, if you would like to get started more quickly, the Laravel application starter kits include support for this feature!

Configuration

After confirming their password, a user will not be asked to confirm their password again for three hours. However, you may configure the length of time before the user is re-prompted for their password by changing the value of the `password_timeout` configuration value within your application's `config/auth.php` configuration file.

Routing

The Password Confirmation Form

First, we will define a route to display a view that requests that the user confirm their password:

```
Route::get('/confirm-password', function () {
    return view('auth.confirm-password');
})->middleware('auth')->name('password.confirm');
```

As you might expect, the view that is returned by this route should have a form containing a `password` field. In addition, feel free to include text within the view that explains that the user is entering a protected area of the application and must confirm their password.

Confirming The Password

Next, we will define a route that will handle the form request from the "confirm password" view. This route will be responsible for validating the password and redirecting the user to their intended destination:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1'])->name('password.confirm');
```

Before moving on, let's examine this route in more detail. First, the request's `password` field is determined to actually match the authenticated user's password. If the password is valid, we need to inform Laravel's session that the user has confirmed their password. The `passwordConfirmed` method will set a timestamp in the user's session that Laravel can use to determine when the user last confirmed their password. Finally, we can redirect the user to their intended destination.

Protecting Routes

You should ensure that any route that performs an action which requires recent password confirmation is assigned the `password.confirm` middleware. This middleware is included with the default installation of Laravel and will automatically store the user's intended destination in the session so that the user may be redirected to that location after confirming their password. After storing the user's intended destination in the session, the middleware will redirect the user to the `password.confirm` named route:

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

Adding Custom Guards

You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place

your call to the `extend` method within a service provider. Since Laravel already ships with an `AuthServiceProvider`, we can place the code in that provider:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::extend('jwt', function ($app, $name, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\Guard...

            return new JwtGuard(Auth::createUserProvider($config['provider']));
        });
    }
}
```

As you can see in the example above, the callback passed to the `extend` method should return an implementation of `Illuminate\Contracts\Auth\Guard`. This interface contains a few methods you will need to implement to define a custom guard. Once your custom guard has been defined, you may reference the guard in the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

Closure Request Guards

The simplest way to implement a custom, HTTP request based authentication system is by using the `Auth::viaRequest` method. This method allows you to quickly define your authentication process using a single closure.

To get started, call the `Auth::viaRequest` method within the `boot` method of your `AuthServiceProvider`. The `viaRequest` method accepts an authentication driver name as its first argument. This name can be any string that describes your custom guard. The second argument passed to the method should be a closure that receives the incoming HTTP request and returns a user instance or, if authentication fails, `null`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Register any application authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Auth::viaRequest('custom-token', function (Request $request) {
        return User::where('token', $request->token)->first();
    });
}
```

Once your custom authentication driver has been defined, you may configure it as a driver within the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

Adding Custom User Providers

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication user provider. We will use the `provider` method on the `Auth` facade to define a custom user provider. The user provider resolver should return an implementation of `Illuminate\Contracts\Auth\UserProvider`:

```
<?php

namespace App\Providers;

use App\Extensions\MongoUserProvider;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::provider('mongo', function ($app, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\UserProvider...

            return new MongoUserProvider($app->make('mongo.connection'));
        });
    }
}
```

After you have registered the provider using the `provider` method, you may switch to the new user provider in your `auth.php` configuration file. First, define a `provider` that uses your new driver:

```
'providers' => [
    'users' => [
        'driver' => 'mongo',
    ],
],
```

Finally, you may reference this provider in your `guards` configuration:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

The User Provider Contract

`Illuminate\Contracts\Auth\UserProvider` implementations are responsible for fetching an `Illuminate\Contracts\Auth\Authenticatable` implementation out of a persistent storage system, such as MySQL, MongoDB, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent the authenticated user:

Let's take a look at the `Illuminate\Contracts\Auth\UserProvider` contract:

```
<?php

namespace Illuminate\Contracts\Auth;
```

```
interface UserProvider
{
    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);
}
```

The `retrieveById` function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The `Authenticatable` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByToken` function retrieves a user by their unique `$identifier` and "remember me" `$token`, typically stored in a database column like `remember_token`. As with the previous method, the `Authenticatable` implementation with a matching token value should be returned by this method.

The `updateRememberToken` method updates the `$user` instance's `remember_token` with the new `$token`. A fresh token is assigned to users on a successful "remember me" authentication attempt or when the user is logging out.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to authenticate with an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition that searches for a user record with a "username" matching the value of `$credentials['username']`. The method should return an implementation of `Authenticatable`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given `$user` with the `$credentials` to authenticate the user. For example, this method will typically use the `Hash::check` method to compare the value of `$user->getAuthPassword()` to the value of `$credentials['password']`. This method should return `true` or `false` indicating whether the password is valid.

The Authenticatable Contract

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable` contract. Remember, user providers should return implementations of this interface from the `retrieveById`, `retrieveByToken`, and `retrieveByCredentials` methods:

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable
{
    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();
}
```

This interface is simple. The `getAuthIdentifierName` method should return the name of the "primary key" field of the user and the `getAuthIdentifier` method should return the "primary key" of the user. When using a MySQL back-end, this would likely be the auto-incrementing primary key assigned to the user record. The `getAuthPassword` method should return the user's hashed password.

This interface allows the authentication system to work with any "user" class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a `App\Models\User` class in the `app/Models` directory which implements this interface.

Events

Laravel dispatches a variety of events during the authentication process. You may attach listeners to these events in your `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [
        'App\Listeners\LogRegisteredUser',
    ],

    'Illuminate\Auth\Events\Attempting' => [
        'App\Listeners\LogAuthenticationAttempt',
    ],

    'Illuminate\Auth\Events\Authenticated' => [
        'App\Listeners\LogAuthenticated',
    ],

    'Illuminate\Auth\Events\Login' => [
        'App\Listeners\LogSuccessfulLogin',
    ],

    'Illuminate\Auth\Events\Failed' => [
        'App\Listeners\LogFailedLogin',
    ],

    'Illuminate\Auth\Events\Validated' => [
        'App\Listeners\LogValidated',
    ],

    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerified',
    ],

    'Illuminate\Auth\Events\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

    'Illuminate\Auth\Events\CurrentDeviceLogout' => [
        'App\Listeners\LogCurrentDeviceLogout',
    ],

    'Illuminate\Auth\Events\OtherDeviceLogout' => [
        'App\Listeners\LogOtherDeviceLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
    ],

    'Illuminate\Auth\Events\PasswordReset' => [
        'App\Listeners\LogPasswordReset',
    ],
];
```

Security

Authorization

- Introduction
- Gates
  - Writing Gates
  - Authorizing Actions
  - Gate Responses
  - Intercepting Gate Checks
- Creating Policies
  - Generating Policies
  - Registering Policies
- Writing Policies
  - Policy Methods
  - Policy Responses
  - Methods Without Models
  - Guest Users
  - Policy Filters
- Authorizing Actions Using Policies
  - Via The User Model
  - Via Controller Helpers
  - Via Middleware
  - Via Blade Templates
  - Supplying Additional Context

Introduction

In addition to providing built-in authentication services, Laravel also provides a simple way to authorize user actions against a given resource. For example, even though a user is authenticated, they may not be authorized to update or delete certain Eloquent models or database records managed by your application. Laravel's authorization features provide an easy, organized way of managing these types of authorization checks.

Laravel provides two primary ways of authorizing actions: gates and policies. Think of gates and policies like routes and controllers. Gates provide a simple, closure based approach to authorization while policies, like controllers, group logic around a particular model or resource. In this documentation, we'll explore gates first and then examine policies.

You do not need to choose between exclusively using gates or exclusively using policies when building an application. Most applications will most likely contain some mixture of gates and policies, and that is perfectly fine! Gates are most applicable to actions which are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Gates

Writing Gates

> NOTE Gates are a great way to learn the basics of Laravel's authorization features; however, when building robust Laravel applications you should consider using policies to organize your authorization rules.

Gates are simply closures that determine if a user is authorized to perform a given action. Typically, gates are defined within the `boot` method of the `App\Providers\AuthServiceProvider` class using the `Gate` facade. Gates always receive a user instance as their first argument and may optionally receive additional arguments such as a relevant Eloquent model.

In this example, we'll define a gate to determine if a user can update a given `App\Models\Post` model. The gate will accomplish this may comparing the user's `id` against the `user_id` of the user that created the post:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

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

Like controllers, gates may also be defined using a class callback array:

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

Authorizing Actions

To authorize an action using gates, you should use the `allows` or `denies` methods provided by the `Gate` facade. Note that you are not required to pass the currently authenticated user to these methods. Laravel will automatically take care of passing the user into the gate closure. It is typical to call the gate authorization methods within your application's controllers before performing an action that requires authorization:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Post $post)
    {
        if (! Gate::allows('update-post', $post)) {
            abort(403);
        }

        // Update the post...
    }
}
```

If you would like to determine if a user other than the currently authenticated user is authorized to perform an action, you may use the `forUser` method on the `Gate` facade:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...
}
```

```
if (Gate::forUser($user)->denies('update-post', $post)) {
    // The user can't update the post...
}
```

You may authorize multiple actions at a time using the `any` or `none` methods:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user can't update or delete the post...
}
```

Authorizing Or Throwing Exceptions

If you would like to attempt to authorize an action and automatically throw an `Illuminate\Auth\Access\AuthorizationException` if the user is not allowed to perform the given action, you may use the `Gate` facade's `authorize` method. Instances of `AuthorizationException` are automatically converted to a 403 HTTP response by Laravel's exception handler:

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

Supplying Additional Context

The gate methods for authorizing abilities (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) and the authorization Blade directives (`@can`, `@cannot`, `@canany`) can receive an array as their second argument. These array elements are passed as parameters to the gate closure, and can be used for additional context when making authorization decisions:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category, $pinned) {
    if (! $user->canPublishToGroup($category->group)) {
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false;
    }

    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
    // The user can create the post...
}
```

Gate Responses

So far, we have only examined gates that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an `Illuminate\Auth\Access\Response` from your gate:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::deny('You must be an administrator.');
});
```

Even when you return an authorization response from your gate, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('edit-settings');

// The action is authorized...
```

Intercepting Gate Checks

Sometimes, you may wish to grant all abilities to a specific user. You may use the `before` method to define a closure that is run before all other authorization checks:

```
use Illuminate\Support\Facades\Gate;

Gate::before(function ($user, $ability) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

If the `before` closure returns a non-null result that result will be considered the result of the authorization check.

You may use the `after` method to define a closure to be executed after all other authorization checks:

```
Gate::after(function ($user, $ability, $result, $arguments) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

Similar to the `before` method, if the `after` closure returns a non-null result that result will be considered the result of the authorization check.

Creating Policies

Generating Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, if your application is a blog, you may have a `App\Models\Post` model and a corresponding `App\Policies\PostPolicy` to authorize user actions such as creating or updating posts.

You may generate a policy using the `make:policy` Artisan command. The generated policy will be placed in the `app/Policies` directory. If this directory does not exist in your application, Laravel will create it for you:

```
php artisan make:policy PostPolicy
```

The `make:policy` command will generate an empty policy class. If you would like to generate a class with example policy methods related to viewing, creating, updating, and deleting the resource, you may provide a `--model` option when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

Registering Policies

Once the policy class has been created, it needs to be registered. Registering policies is how we can inform Laravel which policy to use when authorizing actions against a given model type.

The `App\Providers\AuthServiceProvider` included with fresh Laravel applications contains a `policies` property

which maps your Eloquent models to their corresponding policies. Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given Eloquent model:

```
<?php

namespace App\Providers;

use App\Models\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

Policy Auto-Discovery

Instead of manually registering model policies, Laravel can automatically discover policies as long as the model and policy follow standard Laravel naming conventions. Specifically, the policies must be in a `Policies` directory at or above the directory that contains your models. So, for example, the models may be placed in the `app/Models` directory while the policies may be placed in the `app/Policies` directory. In this situation, Laravel will check for policies in `app/Models/Policies` then `app/Policies`. In addition, the policy name must match the model name and have a `Policy` suffix. So, a `User` model would correspond to a `UserPolicy` policy class.

If you would like to define your own policy discovery logic, you may register a custom policy discovery callback using the `Gate::guessPolicyNamesUsing` method. Typically, this method should be called from the `boot` method of your application's `AuthServiceProvider`:

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function ($modelClass) {
    // Return the name of the policy class for the given model...
});
```

> NOTE Any policies that are explicitly mapped in your `AuthServiceProvider` will take precedence over any potentially auto-discovered policies.

Writing Policies

Policy Methods

Once the policy class has been registered, you may add methods for each action it authorizes. For example, let's define an `update` method on our `PostPolicy` which determines if a given `App\Models\User` can update a given `App\Models\Post` instance.

The `update` method will receive a `User` and a `Post` instance as its arguments, and should return `true` or `false` indicating whether the user is authorized to update the given `Post`. So, in this example, we will verify that the user's `id` matches the `user_id` on the post:

```
<?php

namespace App\Policies;

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

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

You may continue to define additional methods on the policy as needed for the various actions it authorizes. For example, you might define `view` or `delete` methods to authorize various `Post` related actions, but remember you are free to give your policy methods any name you like.

If you used the `--model` option when generating your policy via the Artisan console, it will already contain methods for the `viewAny`, `view`, `create`, `update`, `delete`, `restore`, and `forceDelete` actions.

> TIP All policies are resolved via the Laravel service container, allowing you to type-hint any needed dependencies in the policy's constructor to have them automatically injected.

Policy Responses

So far, we have only examined policy methods that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an `Illuminate\Auth\Access\Response` instance from your policy method:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\Models\User  $user
 * @param  \App\Models\Post  $post
 * @return \Illuminate\Auth\Access\Response
 */
public function update(User $user, Post $post)
{
    return $user->id === $post->user_id
                ? Response::allow()
                : Response::deny('You do not own this post.');
}
```

When returning an authorization response from your policy, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
use Illuminate\Support\Facades\Gate;

$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('update', $post);

// The action is authorized...
```

Methods Without Models

Some policy methods only receive an instance of the currently authenticated user. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to determine if a user is authorized to create any posts at all. In these situations, your policy method should only expect to receive a user instance:

```
/**
 * Determine if the given user can create posts.
 *
 * @param  \App\Models\User  $user
 * @return bool
 */
public function create(User $user)
{
    return $user->role == 'writer';
}
```

Guest Users

By default, all gates and policies automatically return `false` if the incoming HTTP request was not initiated by an authenticated user. However, you may allow these authorization checks to pass through to your gates and policies by declaring an "optional" type-hint or supplying a `null` default value for the user argument definition:

```
<?php

namespace App\Policies;

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

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function update(?User $user, Post $post)
    {
        return optional($user)->id === $post->user_id;
    }
}
```

Policy Filters

For certain users, you may wish to authorize all actions within a given policy. To accomplish this, define a `before` method on the policy. The `before` method will be executed before any other methods on the policy, giving you an opportunity to authorize the action before the intended policy method is actually called. This feature is most commonly used for authorizing application administrators to perform any action:

```
use App\Models\User;

/**
 * Perform pre-authorization checks.
 *
 * @param  \App\Models\User  $user
 * @param  string  $ability
 * @return void|bool
 */
public function before(User $user, $ability)
{
    if ($user->isAdministrator()) {
        return true;
    }
}
```

If you would like to deny all authorization checks for a particular type of user then you may return `false` from the `before` method. If `null` is returned, the authorization check will fall through to the policy method.

> NOTE The `before` method of a policy class will not be called if the class doesn't contain a method with a name matching the name of the ability being checked.

Authorizing Actions Using Policies

Via The User Model

The `App\Models\User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cannot`. The `can` and `cannot` methods receive the name of the action you wish to authorize and the relevant model. For example, let's determine if a user is authorized to update a given `App\Models\Post` model. Typically, this will be done within a controller method:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Post $post)
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }

        // Update the post...
    }
}
```

If a policy is registered for the given model, the `can` method will automatically call the appropriate policy and return the boolean result. If no policy is registered for the model, the `can` method will attempt to call the closure based Gate matching the given action name.

Actions That Don't Require Models

Remember, some actions may correspond to policy methods like `create` that do not require a model instance. In these situations, you may pass a class name to the `can` method. The class name will be used to determine which policy to use when authorizing the action:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Create a post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        if ($request->user()->cannot('create', Post::class)) {
```

```
            abort(403);
        }

        // Create the post...
    }
}
```

Via Controller Helpers

In addition to helpful methods provided to the `App\Models\User` model, Laravel provides a helpful `authorize` method to any of your controllers which extend the `App\Http\Controllers\Controller` base class.

Like the `can` method, this method accepts the name of the action you wish to authorize and the relevant model. If the action is not authorized, the `authorize` method will throw an `Illuminate\Auth\Access\AuthorizationException` exception which the Laravel exception handler will automatically convert to an HTTP response with a 403 status code:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post)
    {
        $this->authorize('update', $post);

        // The current user can update the blog post...
    }
}
```

Actions That Don't Require Models

As previously discussed, some policy methods like `create` do not require a model instance. In these situations, you should pass a class name to the `authorize` method. The class name will be used to determine which policy to use when authorizing the action:

```
use App\Models\Post;
use Illuminate\Http\Request;

/**
 * Create a new blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}
```

Authorizing Resource Controllers

If you are utilizing resource controllers, you may make use of the `authorizeResource` method in your controller's constructor. This method will attach the appropriate `can` middleware definitions to the resource controller's

methods.

The `authorizeResource` method accepts the model's class name as its first argument, and the name of the route / request parameter that will contain the model's ID as its second argument. You should ensure your resource controller is created using the `--model` flag so that it has the required method signatures and type hints:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Create the controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

The following controller methods will be mapped to their corresponding policy method. When requests are routed to the given controller method, the corresponding policy method will automatically be invoked before the controller method is executed:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

> TIP You may use the `make:policy` command with the `--model` option to quickly generate a policy class for a given model: `php artisan make:policy PostPolicy --model=Post`.

Via Middleware

Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers. By default, the `Illuminate\Auth\Middleware\Authorize` middleware is assigned the `can` key in your `App\Http\Kernel` class. Let's explore an example of using the `can` middleware to authorize that a user can update a post:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

In this example, we're passing the `can` middleware two arguments. The first is the name of the action we wish to authorize and the second is the route parameter we wish to pass to the policy method. In this case, since we are using implicit model binding, a `App\Models\Post` model will be passed to the policy method. If the user is not authorized to perform the given action, an HTTP response with a 403 status code will be returned by the middleware.

Actions That Don't Require Models

Again, some policy methods like `create` do not require a model instance. In these situations, you may pass a class name to the middleware. The class name will be used to determine which policy to use when authorizing

the action:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Models\Post');
```

Via Blade Templates

When writing Blade templates, you may wish to display a portion of the page only if the user is authorized to perform a given action. For example, you may wish to show an update form for a blog post only if the user can actually update the post. In this situation, you may use the `@can` and `@cannot` directives:

```
@can('update', $post)
    <!-- The current user can update the post... -->
@elsecan('create', App\Models\Post::class)
    <!-- The current user can create new posts... -->
@endcan

@cannot('update', $post)
    <!-- The current user cannot update the post... -->
@elsecannot('create', App\Models\Post::class)
    <!-- The current user can now create new posts... -->
@endcannot
```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above are equivalent to the following statements:

```
@if (Auth::user()->can('update', $post))
    <!-- The current user can update the post... -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- The current user cannot update the post... -->
@endunless
```

You may also determine if a user is authorized to perform any action from a given array of actions. To accomplish this, use the `@canany` directive:

```
@canany(['update', 'view', 'delete'], $post)
    <!-- The current user can update, view, or delete the post... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- The current user can create a post... -->
@endcanany
```

Actions That Don't Require Models

Like most of the other authorization methods, you may pass a class name to the `@can` and `@cannot` directives if the action does not require a model instance:

```
@can('create', App\Models\Post::class)
    <!-- The current user can create posts... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- The current user can't create posts... -->
@endcannot
```

Supplying Additional Context

When authorizing actions using policies, you may pass an array as the second argument to the various authorization functions and helpers. The first element in the array will be used to determine which policy should be invoked, while the rest of the array elements are passed as parameters to the policy method and can be used for additional context when making authorization decisions. For example, consider the following `PostPolicy` method definition which contains an additional `$category` parameter:

```
/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\Models\User  $user
 * @param  \App\Models\  $post
```

```
 * @param  int  $category
 * @return bool
 */
public function update(User $user, Post $post, int $category)
{
    return $user->id === $post->user_id &&
           $user->canUpdateCategory($category);
}
```

When attempting to determine if the authenticated user can update a given post, we can invoke this policy method like so:

```
/**
 * Update the given blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \App\Models\Post  $post
 * @return \Illuminate\Http\Response
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post)
{
    $this->authorize('update', [$post, $request->category]);

    // The current user can update the blog post...
}
```

Security

Email Verification

- Introduction
  - Model Preparation
  - Database Preparation
- Routing
  - The Email Verification Notice
  - The Email Verification Handler
  - Resending The Verification Email
  - Protecting Routes
- Customization
- Events

Introduction

Many web applications require users to verify their email addresses before using the application. Rather than forcing you to re-implement this feature by hand for each application you create, Laravel provides convenient built-in services for sending and verifying email verification requests.

> TIP Want to get started fast? Install one of the Laravel application starter kits in a fresh Laravel application. The starter kits will take care of scaffolding your entire authentication system, including email verification support.

Model Preparation

Before getting started, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\MustVerifyEmail` contract:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

Once this interface has been added to your model, newly registered users will automatically be sent an email containing an email verification link. As you can see by examining your application's `App\Providers\EventServiceProvider`, Laravel already contains a `SendEmailVerificationNotification` listener that is attached to the `Illuminate\Auth\Events\Registered` event. This event listener will send the email verification link to the user.

If you are manually implementing registration within your application instead of using a starter kit, you should ensure that you are dispatching the `Illuminate\Auth\Events\Registered` event after a user's registration is successful:

```
use Illuminate\Auth\Events\Registered;

event(new Registered($user));
```

Database Preparation

Next, your `user` table must contain an `email_verified_at` column to store the date and time that the user's email address was verified. By default, the `users` table migration included with the Laravel framework already

includes this column. So, all you need to do is run your database migrations:

```
php artisan migrate
```

Routing

To properly implement email verification, three routes will need to be defined. First, a route will be needed to display a notice to the user that they should click the email verification link in the verification email that Laravel sent them after registration.

Second, a route will be needed to handle requests generated when the user clicks the email verification link in the email.

Third, a route will be needed to resend a verification link if the user accidentally loses the first verification link.

The Email Verification Notice

As mentioned previously, a route should be defined that will return a view instructing the user to click the email verification link that was emailed to them by Laravel after registration. This view will be displayed to users when they try to access other parts of the application without verifying their email address first. Remember, the link is automatically emailed to the user as long as your `App\Models\User` model implements the `MustVerifyEmail` interface:

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

The route that returns the email verification notice should be named `verification.notice`. It is important that the route be assigned this exact name since the `verified` middleware included with Laravel will automatically redirect to this route name if a user has not verified their email address.

> TIP When manually implementing email verification, you are required to define the contents of the verification notice view yourself. If you would like scaffolding that includes all necessary authentication and verification views, check out the Laravel application starter kits.

The Email Verification Handler

Next, we need to define a route that will handle requests generated when the user clicks the email verification link that was emailed to them. This route should be named `verification.verify` and be assigned the `auth` and `signed` middlewares:

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;
use Illuminate\Http\Request;

Route::get('/email/verify/{id}/{hash}', function (EmailVerificationRequest $request) {
    $request->fulfill();

    return redirect('/home');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

Before moving on, let's take a closer look at this route. First, you'll notice we are using an `EmailVerificationRequest` request type instead of the typical `Illuminate\Http\Request` instance. The `EmailVerificationRequest` is a form request that is included with Laravel. This request will automatically take care of validating the request's `id` and `hash` parameters.

Next, we can proceed directly to calling the `fulfill` method on the request. This method will call the `markEmailAsVerified` method on the authenticated user and dispatch the `Illuminate\Auth\Events\Verified` event. The `markEmailAsVerified` method is available to the default `App\Models\User` model via the `Illuminate\Foundation\Auth\User` base class. Once the user's email address has been verified, you may redirect them wherever you wish.

Resending The Verification Email

Sometimes a user may misplace or accidentally delete the email address verification email. To accommodate this, you may wish to define a route to allow the user to request that the verification email be resent. You may then make a request to this route by placing a simple form submission button within your verification notice view:

```
use Illuminate\Http\Request;

Route::post('/email/verification-notification', function (Request $request) {
    $request->user()->sendEmailVerificationNotification();

    return back()->with('message', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
```

Protecting Routes

Route middleware may be used to only allow verified users to access a given route. Laravel ships with a `verified` middleware, which references the `Illuminate\Auth\Middleware\EnsureEmailIsVerified` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('/profile', function () {
    // Only verified users may access this route...
})->middleware('verified');
```

If an unverified user attempts to access a route that has been assigned this middleware, they will automatically be redirected to the `verification.notice` named route.

Customization

Verification Email Customization

Although the default email verification notification should satisfy the requirements of most applications, Laravel allows you to customize how the email verification mail message is constructed.

To get started, pass a closure to the `toMailUsing` method provided by the `Illuminate\Auth\Notifications\VerifyEmail` notification. The closure will receive the notifiable model instance that is receiving the notification as well as the signed email verification URL that the user must visit to verify their email address. The closure should return an instance of `Illuminate\Notifications\Messages\MailMessage`. Typically, you should call the `toMailUsing` method from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    // ...

    VerifyEmail::toMailUsing(function ($notifiable, $url) {
        return (new MailMessage)
            ->subject('Verify Email Address')
            ->line('Click the button below to verify your email address.')
            ->action('Verify Email Address', $url)
    });
}
```

TIP To learn more about mail notifications, please consult the mail notification documentation.

Events

When using the Laravel application starter kits, Laravel dispatches events during the email verification process. If you are manually handling email verification for your application, you may wish to manually dispatch these

events after verification is completed. You may attach listeners to these events in your application's `EventServiceProvider`:

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

Security

Encryption

- Introduction
- Configuration
- Using The Encrypter

Introduction

Laravel's encryption services provide a simple, convenient interface for encrypting and decrypting text via OpenSSL using AES-256 and AES-128 encryption. All of Laravel's encrypted values are signed using a message authentication code (MAC) so that their underlying value can not be modified or tampered with once encrypted.

Configuration

Before using Laravel's encrypter, you must set the `key` configuration option in your `config/app.php` configuration file. This configuration value is driven by the `APP_KEY` environment variable. You should use the `php artisan key:generate` command to generate this variable's value since the `key:generate` command will use PHP's secure random bytes generator to build a cryptographically secure key for your application. Typically, the value of the `APP_KEY` environment variable will be generated for you during Laravel's installation.

Using The Encrypter

Encrypting A Value

You may encrypt a value using the `encryptString` method provided by the `Crypt` facade. All encrypted values are encrypted using OpenSSL and the AES-256-CBC cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC). The integrated message authentication code will prevent the decryption of any values that have been tampered with by malicious users:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class DigitalOceanTokenController extends Controller
{
    /**
     * Store a DigitalOcean API token for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function storeSecret(Request $request)
    {
        $request->user()->fill([
            'token' => Crypt::encryptString($request->token),
        ])->save();
    }
}
```

Decrypting A Value

You may decrypt values using the `decryptString` method provided by the `Crypt` facade. If the value can not be properly decrypted, such as when the message authentication code is invalid, an `Illuminate\Contracts\Encryption\DecryptException` will be thrown:

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

```
use Illuminate\Support\Facades\Crypt;

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

Security

Hashing

- Introduction
- Configuration
- Basic Usage
  - Hashing Passwords
  - Verifying That A Password Matches A Hash
  - Determining If A Password Needs To Be Rehashed

Introduction

The Laravel `Hash` facade provides secure Bcrypt and Argon2 hashing for storing user passwords. If you are using one of the Laravel application starter kits, Bcrypt will be used for registration and authentication by default.

Bcrypt is a great choice for hashing passwords because its "work factor" is adjustable, which means that the time it takes to generate a hash can be increased as hardware power increases. When hashing passwords, slow is good. The longer an algorithm takes to hash a password, the longer it takes malicious users to generate "rainbow tables" of all possible string hash values that may be used in brute force attacks against applications.

Configuration

The default hashing driver for your application is configured in your application's `config/hashing.php` configuration file. There are currently several supported drivers: Bcrypt and Argon2 (Argon2i and Argon2id variants).

> NOTE The Argon2i driver requires PHP 7.2.0 or greater and the Argon2id driver requires PHP 7.3.0 or greater.

Basic Usage

Hashing Passwords

You may hash a password by calling the `make` method on the `Hash` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class PasswordController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        // Validate the new password length...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();
    }
}
```

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,
]);
```

> TIP For more information on these options, please refer to the official PHP documentation regarding Argon hashing.

Verifying That A Password Matches A Hash

The `check` method provided by the `Hash` facade allows you to verify that a given plain-text string corresponds to a given hash:

```
if (Hash::check('plain-text', $hashedPassword)) {
    // The passwords match...
}
```

Determining If A Password Needs To Be Rehashed

The `needsRehash` method provided by the `Hash` facade allows you to determine if the work factor used by the hasher has changed since the password was hashed. Some applications choose to perform this check during the application's authentication process:

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

Security

Resetting Passwords

- Introduction
  - Model Preparation
  - Database Preparation
- Routing
  - Requesting The Password Reset Link
  - Resetting The Password
- Customization

Introduction

Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this by hand for every application you create, Laravel provides convenient services for sending password reset links and secure resetting passwords.

> TIP Want to get started fast? Install a Laravel application starter kit in a fresh Laravel application. Laravel's starter kits will take care of scaffolding your entire authentication system, including resetting forgotten passwords.

Model Preparation

Before using the password reset features of Laravel, your application's `App\Models\User` model must use the `Illuminate\Notifications\Notifiable` trait. Typically, this trait is already included on the default `App\Models\User` model that is created with new Laravel applications.

Next, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. The `App\Models\User` model included with the framework already implements this interface, and uses the `Illuminate\Auth\Passwords\CanResetPassword` trait to include the methods needed to implement the interface.

Database Preparation

A table must be created to store your application's password reset tokens. The migration for this table is included in the default Laravel application, so you only need to migrate your database to create this table:

```
php artisan migrate
```

Routing

To properly implement support for allowing users to reset their passwords, we will need to define several routes. First, we will need a pair of routes to handle allowing the user to request a password reset link via their email address. Second, we will need a pair of routes to handle actually resetting the password once the user visits the password reset link that is emailed to them and completes the password reset form.

Requesting The Password Reset Link

The Password Reset Link Request Form

First, we will define the routes that are needed to request password reset links. To get started, we will define a route that returns a view with the password reset link request form:

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

The view that is returned by this route should have a form containing an `email` field, which will allow the user to request a password reset link for a given email address.

Handling The Form Submission

Next, we will define a route that handles the form submission request from the "forgot password" view. This route will be responsible for validating the email address and sending the password reset request to the corresponding user:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

Route::post('/forgot-password', function (Request $request) {
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
                ? back()->with(['status' => __($status)])
                : back()->withErrors(['email' => __($status)]);
})->middleware('guest')->name('password.email');
```

Before moving on, let's examine this route in more detail. First, the request's `email` attribute is validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to send a password reset link to the user. The password broker will take care of retrieving the user by the given field (in this case, the email address) and sending the user a password reset link via Laravel's built-in notification system.

The `sendResetLink` method returns a "status" slug. This status may be translated using Laravel's localization helpers in order to display a user-friendly message to the user regarding the status of their request. The translation of the password reset status is determined by your application's `resources/lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file.

You may wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `sendResetLink` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the authentication documentation

> TIP When manually implementing password resets, you are required to define the contents of the views and routes yourself. If you would like scaffolding that includes all necessary authentication and verification logic, check out the Laravel application starter kits.

Resetting The Password

The Password Reset Form

Next, we will define the routes necessary to actually reset the password once the user clicks on the password reset link that has been emailed to them and provides a new password. First, let's define the route that will display the reset password form that is displayed when the user clicks the reset password link. This route will receive a `token` parameter that we will use later to verify the password reset request:

```
Route::get('/reset-password/{token}', function ($token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

The view that is returned by this route should display a form containing an `email` field, a `password` field, a `password_confirmation` field, and a hidden `token` field, which should contain the value of the secret `$token` received by our route.

Handling The Form Submission

Of course, we need to define a route to actually handle the password reset form submission. This route will be responsible for validating the incoming request and updating the user's password in the database:

```
use 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, $password) use ($request) {
            $user->forceFill([
                'password' => Hash::make($password)
            ])->save();

            $user->setRememberToken(Str::random(60));

            event(new PasswordReset($user));
        }
    );

    return $status == Password::PASSWORD_RESET
                ? redirect()->route('login')->with('status', __($status))
                : back()->withErrors(['email' => [__($status)]]);
})->middleware('guest')->name('password.update');
```

Before moving on, let's examine this route in more detail. First, the request's `token`, `email`, and `password` attributes are validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to validate the password reset request credentials.

If the token, email address, and password given to the password broker are valid, the closure passed to the `reset` method will be invoked. Within this closure, which receives the user instance and the plain-text password provided to the password reset form, we may update the user's password in the database.

The `reset` method returns a "status" slug. This status may be translated using Laravel's localization helpers in order to display a user-friendly message to the user regarding the status of their request. The translation of the password reset status is determined by your application's `resources/lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file.

Before moving on, you may wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `reset` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the authentication documentation

Customization

Reset Link Customization

You may customize the password reset link URL using the `createUrlUsing` method provided by the `ResetPassword` notification class. This method accepts a closure which receives the user instance that is receiving the notification as well as the password reset link token. Typically, you should call this method from your `App\Providers\AuthServiceProvider` service provider's `boot` method:

```
use Illuminate\Auth\Notifications\ResetPassword;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    ResetPassword::createUrlUsing(function ($user, string $token) {
        return 'https://example.com/reset-password?token='.$token;
```

```
    });
}
```

Reset Email Customization

You may easily modify the notification class used to send the password reset link to the user. To get started, override the `sendPasswordResetNotification` method on your `App\Models\User` model. Within this method, you may send the notification using any notification class of your own creation. The password reset `$token` is the first argument received by the method. You may use this `$token` to build the password reset URL of your choice and send your notification to the user:

```
use App\Notifications\ResetPasswordNotification;

/**
 * Send a password reset notification to the user.
 *
 * @param  string  $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $url = 'https://example.com/reset-password?token='.$token;

    $this->notify(new ResetPasswordNotification($url));
}
```

Database

Database: Getting Started

- Introduction
  - Configuration
  - Read & Write Connections
- Running SQL Queries
  - Using Multiple Database Connections
  - Listening For Query Events
- Database Transactions
- Connecting To The Database CLI

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 four databases:

- MySQL 5.6+ (Version Policy)
- PostgreSQL 9.4+ (Version Policy)
- SQLite 3.8.8+
- SQL Server 2017+ (Version Policy)

Configuration

The configuration for Laravel's database services is located in your application's `config/database.php` configuration file. In this file you may define all of your database connections, as well as specify which connection should be used by default. Most of the configuration options within this file are driven by the values of your application's environment variables. Examples for most of Laravel's supported database systems are provided in this file.

By default, Laravel's sample environment configuration is ready to use with Laravel Sail, which is a Docker configuration for developing Laravel applications on your local machine. However, you are free to modify your database configuration as needed for your local database.

SQLite Configuration

SQLite databases are contained within a single file on your filesystem. You can create a new SQLite database using the `touch` command in your terminal: `touch database/database.sqlite`. After the database has been created, you may easily configure your environment variables to point to this database by placing the absolute path to the database in the `DB_DATABASE` environment variable:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

To enable foreign key constraints for SQLite connections, you should set the `DB_FOREIGN_KEYS` environment variable to `true`:

```
DB_FOREIGN_KEYS=true
```

Microsoft SQL Server Configuration

To use a Microsoft SQL Server database, you should ensure that you have the `sqlsrv` and `pdo_sqlsrv` PHP extensions installed as well as any dependencies they may require such as the Microsoft SQL ODBC driver.

Configuration Using URLs

Typically, database connections are configured using multiple configuration values such as `host`, `database`,

`username`, `password`, etc. Each of these configuration values has its own corresponding environment variable. This means that when configuring your database connection information on a production server, you need to manage several environment variables.

Some managed database providers such as AWS and Heroku provide a single database "URL" that contains all of the connection information for the database in a single string. An example database URL may look something like the following:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

These URLs typically follow a standard schema convention:

```
driver://username:password@host:port/database?options
```

For convenience, Laravel supports these URLs as an alternative to configuring your database with multiple configuration options. If the `url` (or corresponding `DATABASE_URL` environment variable) configuration option is present, it will be used to extract the database connection and credential information.

Read & Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

Note that three keys have been added to the configuration array: `read`, `write` and `sticky`. The `read` and `write` keys have array values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` configuration array.

You only need to place items in the `read` and `write` arrays if you wish to override the values from the main `mysql` array. So, in this case, `192.168.1.1` will be used as the host for the "read" connection, while `192.168.1.3` will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` array will be shared across both connections. When multiple values exist in the `host` configuration array, a database host will be randomly chosen for each request.

The `sticky` Option

The `sticky` option is an *optional* value that can be used to allow the immediate reading of records that have been written to the database during the current request cycle. If the `sticky` option is enabled and a "write" operation has been performed against the database during the current request cycle, any further "read" operations will use the "write" connection. This ensures that any data written during the request cycle can be immediately read back from the database during that same request. It is up to you to decide if this is the desired behavior for your application.

Running SQL Queries

Once you have configured your database connection, you may run queries using the `DB` facade. The `DB` facade provides methods for each type of query: `select`, `update`, `insert`, `delete`, and `statement`.

Running A Select Query

To run a basic SELECT query, you may use the `select` method on the `DB` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $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;
}
```

Using Named Bindings

Instead of using `?` to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Running An Insert Statement

To execute an `insert` statement, you may use the `insert` method on the `DB` facade. Like `select`, this method accepts the SQL query as its first argument and bindings as its second argument:

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

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

Running An Update Statement

The `update` method should be used to update existing records in the database. The number of rows affected by the statement is returned by the method:

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

$affected = DB::update(
```

```
    'update users set votes = 100 where name = ?',
    ['Anita']
);
```

Running A Delete Statement

The `delete` method should be used to delete records from the database. Like `update`, the number of rows affected will be returned by the method:

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

$deleted = DB::delete('delete from users');
```

Running A General Statement

Some database statements do not return any value. For these types of operations, you may use the `statement` method on the `DB` facade:

```
DB::statement('drop table users');
```

Running An Unprepared Statement

Sometimes you may want to execute an SQL statement without binding any values. You may use the `DB` facade's `unprepared` method to accomplish this:

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

> NOTE Since unprepared statements do not bind parameters, they may be vulnerable to SQL injection. You should never allow user controlled values within an unprepared statement.

Implicit Commits

When using the `DB` facade's `statement` and `unprepared` methods within transactions you must be careful to avoid statements that cause implicit commits. These statements will cause the database engine to indirectly commit the entire transaction, leaving Laravel unaware of the database's transaction level. An example of such a statement is creating a database table:

```
DB::unprepared('create table a (col varchar(1) null)');
```

Please refer to the MySQL manual for a list of all statements that trigger implicit commits.

Using Multiple Database Connections

If your application defines multiple connections in your `config/database.php` configuration file, you may access each connection via the `connection` method provided by the `DB` facade. The connection name passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file or configured at runtime using the `config` helper:

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

$users = DB::connection('sqlite')->select(...);
```

You may access the raw, underlying PDO instance of a connection using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Listening For Query Events

If you would like to specify a closure that is invoked for each SQL query executed by your application, you may use the `DB` facade's `listen` method. This method can be useful for logging queries or debugging. You may register your query listener closure in the `boot` method of a service provider:

```
<?php
```

```
namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql
            // $query->bindings
            // $query->time
        });
    }
}
```

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. If the closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the `transaction` method:

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

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
});
```

Handling Deadlocks

The `transaction` method accepts an optional second argument which defines the number of times a transaction should be retried when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

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

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the `beginTransaction` method provided by the `DB` facade:

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

DB::beginTransaction();
```

You can rollback the transaction via the `rollBack` method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the `commit` method:

```
DB::commit();
```

> TIP The `DB` facade's transaction methods control the transactions for both the query builder and Eloquent ORM.

Connecting To The Database CLI

If you would like to connect to your database's CLI, you may use the `db` Artisan command:

```
php artisan db
```

If needed, you may specify a database connection name to connect to a database connection that is not the default connection:

```
php artisan db mysql
```

Database

Database: Query Builder

- Introduction
- Running Database Queries
  - Chunking Results
  - Aggregates
- Select Statements
- Raw Expressions
- Joins
- Unions
- Basic Where Clauses
  - Where Clauses
  - Or Where Clauses
  - JSON Where Clauses
  - Additional Where Clauses
  - Logical Grouping
- Advanced Where Clauses
  - Where Exists Clauses
  - Subquery Where Clauses
- Ordering, Grouping, Limit & Offset
  - Ordering
  - Grouping
  - Limit & Offset
- Conditional Clauses
- Insert Statements
  - Upserts
- Update Statements
  - Updating JSON Columns
  - Increment & Decrement
- Delete Statements
- Pessimistic Locking
- Debugging

Introduction

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works perfectly with all of Laravel's supported database systems.

The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings.

> NOTE PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns.

Running Database Queries

Retrieving All Rows From A Table

You may use the `table` method provided by the `DB` facade to begin a query. The `table` method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally retrieve the results of the query using the `get` method:

```
<?php

namespace App\Http\Controllers;

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

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

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

The `get` method returns an `Illuminate\Support\Collection` containing the results of the query where each result is an instance of the PHP `stdClass` object. You may access each column's value by accessing the column as a property of the object:

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

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

TIP Laravel collections provide a variety of extremely powerful methods for mapping and reducing data. For more information on Laravel colletions, check out the collection documentation.

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from a database table, you may use the `DB` facade's `first` method. This method will return a single `stdClass` object:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

If you don't need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

To retrieve a single row by its `id` column value, use the `find` method:

```
$user = DB::table('users')->find(3);
```

Retrieving A List Of Column Values

If you would like to retrieve an `Illuminate\Support\Collection` instance containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve a collection of role titles:

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

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may specify the column that the resulting collection should use as its keys by providing a second argument to the `pluck` method:

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

Chunking Results

If you need to work with thousands of database records, consider using the `chunk` method provided by the `DB` facade. This method retrieves a small chunk of results at a time and feeds each chunk into a closure for processing. For example, let's retrieve the entire `users` table in chunks of 100 records at a time:

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

DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

You may stop further chunks from being processed by returning `false` from the closure:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Process the records...

    return false;
});
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. If you plan to update the retrieved records while chunking, it is always best to use the `chunkById` method instead. This method will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

NOTE When updating or deleting records inside the chunk callback, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the chunked results.

Aggregates

The query builder also provides a variety of methods for retrieving aggregate values like `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

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

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses to fine-tune how your aggregate value is calculated:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Determining If Records Exist

Instead of using the `count` method to determine if any records exist that match your query's constraints, you may use the `exists` and `doesntExist` methods:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

Select Statements

Specifying A Select Clause

You may not always want to select all columns from a database table. Using the `select` method, you can specify a custom "select" clause for the query:

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

$users = DB::table('users')
            ->select('name', 'email as user_email')
            ->get();
```

The `distinct` method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the `addSelect` method:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Raw Expressions

Sometimes you may need to insert an arbitrary string into a query. To create a raw string expression, you may use the `raw` method provided by the `DB` facade:

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count, status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
             ->get();
```

> NOTE Raw statements will be injected into the query as strings, so you should be extremely careful to avoid creating SQL injection vulnerabilities.

Raw Methods

Instead of using the `DB::raw` method, you may also use the following methods to insert a raw expression into various parts of your query. **Remember, Laravel can not guarantee that any query using raw expressions is protected against SQL injection vulnerabilities.**

`selectRaw`

The `selectRaw` method can be used in place of `addSelect(DB::raw(...))`. This method accepts an optional array of bindings as its second argument:

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

`whereRaw / orWhereRaw`

The `whereRaw` and `orWhereRaw` methods can be used to inject a raw "where" clause into your query. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
                ->get();
```

`havingRaw / orHavingRaw`

The `havingRaw` and `orHavingRaw` methods may be used to provide a raw string as the value of the "having"

clause. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

`orderByRaw`

The `orderByRaw` method may be used to provide a raw string as the value of the "order by" clause:

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

`groupByRaw`

The `groupByRaw` method may be used to provide a raw string as the value of the `group by` clause:

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

Joins

Inner Join Clause

The query builder may also be used to add join clauses to your queries. To perform a basic "inner join", you may use the `join` method on a query builder instance. The first argument passed to the `join` method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. You may even join to multiple tables in a single query:

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

$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join / Right Join Clause

If you would like to perform a "left join" or "right join" instead of an "inner join", use the `leftJoin` or `rightJoin` methods. These methods have the same signature as the `join` method:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();

$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Cross Join Clause

You may use the `crossJoin` method to perform a "cross join". Cross joins generate a cartesian product between the first table and the joined table:

```
$sizes = DB::table('sizes')
            ->crossJoin('colors')
            ->get();
```

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a closure as the second argument to the

`join` method. The closure will receive a `Illuminate\Database\Query\JoinClause` instance which allows you to specify constraints on the "join" clause:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

If you would like to use a "where" clause on your joins, you may use the `where` and `orWhere` methods provided by the `JoinClause` instance. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Subquery Joins

You may use the `joinSub`, `leftJoinSub`, and `rightJoinSub` methods to join a query to a subquery. Each of these methods receive three arguments: the subquery, its table alias, and a closure that defines the related columns. In this example, we will retrieve a collection of users where each user record also contains the `created_at` timestamp of the user's most recently published blog post:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
                   ->where('is_published', true)
                   ->groupBy('user_id');

$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function ($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

Unions

The query builder also provides a convenient method to "union" two or more queries together. For example, you may create an initial query and use the `union` method to union it with more queries:

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

$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

In addition to the `union` method, the query builder provides a `unionAll` method. Queries that are combined using the `unionAll` method will not have their duplicate results removed. The `unionAll` method has the same method signature as the `union` method.

Basic Where Clauses

Where Clauses

You may use the query builder's `where` method to add "where" clauses to the query. The most basic call to the `where` method requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to compare against the column's value.

For example, the following query retrieves users where the value of the `votes` column is equal to `100` and the value of the `age` column is greater than `35`:

```
$users = DB::table('users')
                ->where('votes', '=', 100)
                ->where('age', '>', 35)
                ->get();
```

For convenience, if you want to verify that a column is `=` to a given value, you may pass the value as the second argument to the `where` method. Laravel will assume you would like to use the `=` operator:

```
$users = DB::table('users')->where('votes', 100)->get();
```

As previously mentioned, you may use any operator that is supported by your database system:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

You may also pass an array of conditions to the `where` function. Each element of the array should be an array containing the three arguments typically passed to the `where` method:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

Or Where Clauses

When chaining together calls to the query builder's `where` method, the "where" clauses will be joined together using the `and` operator. However, you may use the `orWhere` method to join a clause to the query using the `or` operator. The `orWhere` method accepts the same arguments as the `where` method:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

If you need to group an "or" condition within parentheses, you may pass a closure as the first argument to the `orWhere` method:

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function($query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

The example above will produce the following SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

> NOTE You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

JSON Where Clauses

Laravel also supports querying JSON column types on databases that provide support for JSON column types. Currently, this includes MySQL 5.7+, PostgreSQL, SQL Server 2016, and SQLite 3.9.0 (with the JSON1 extension). To query a JSON column, use the `->` operator:

```
$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

You may use `whereJsonContains` to query JSON arrays. This feature is not supported by the SQLite database:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

If your application uses the MySQL or PostgreSQL databases, you may pass an array of values to the `whereJsonContains` method:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

You may use `whereJsonLength` method to query JSON arrays by their length:

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

Additional Where Clauses

whereBetween / orWhereBetween

The `whereBetween` method verifies that a column's value is between two values:

```
$users = DB::table('users')
           ->whereBetween('votes', [1, 100])
           ->get();
```

whereNotBetween / orWhereNotBetween

The `whereNotBetween` method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

The `whereNotIn` method verifies that the given column's value is not contained in the given array:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

> NOTE If you are adding a large array of integer bindings to your query, the `whereIntegerInRaw` or `whereIntegerNotInRaw` methods may be used to greatly reduce your memory usage.

whereNull / whereNotNull / orWhereNull / orWhereNotNull

The `whereNull` method verifies that the value of the given column is `NULL`:

```
$users = DB::table('users')
                ->whereNull('updated_at')
                ->get();
```

The `whereNotNull` method verifies that the column's value is not `NULL`:

```
$users = DB::table('users')
                ->whereNotNull('updated_at')
                ->get();
```

whereDate / whereMonth / whereDay / whereYear / whereTime

The `whereDate` method may be used to compare a column's value against a date:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

The `whereMonth` method may be used to compare a column's value against a specific month:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

The `whereDay` method may be used to compare a column's value against a specific day of the month:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

The `whereYear` method may be used to compare a column's value against a specific year:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

The `whereTime` method may be used to compare a column's value against a specific time:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

whereColumn / orWhereColumn

The `whereColumn` method may be used to verify that two columns are equal:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
                ->get();
```

You may also pass a comparison operator to the `whereColumn` method:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

You may also pass an array of column comparisons to the `whereColumn` method. These conditions will be joined using the `and` operator:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at'],
                ])->get();
```

Logical Grouping

Sometimes you may need to group several "where" clauses within parentheses in order to achieve your query's desired logical grouping. In fact, you should generally always group calls to the `orWhere` method in parentheses in order to avoid unexpected query behavior. To accomplish this, you may pass a closure to the `where` method:

```
$users = DB::table('users')
           ->where('name', '=', 'John')
           ->where(function ($query) {
               $query->where('votes', '>', 100)
                     ->orWhere('title', '=', 'Admin');
           })
           ->get();
```

As you can see, passing a closure into the `where` method instructs the query builder to begin a constraint group. The closure will receive a query builder instance which you can use to set the constraints that should be

contained within the parenthesis group. The example above will produce the following SQL:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

> NOTE You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

Advanced Where Clauses

Where Exists Clauses

The `whereExists` method allows you to write "where exists" SQL clauses. The `whereExists` method accepts a closure which will receive a query builder instance, allowing you to define the query that should be placed inside of the "exists" clause:

```
$users = DB::table('users')
           ->whereExists(function ($query) {
               $query->select(DB::raw(1))
                     ->from('orders')
                     ->whereColumn('orders.user_id', 'users.id');
           })
           ->get();
```

The query above will produce the following SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

Subquery Where Clauses

Sometimes you may need to construct a "where" clause that compares the results of a subquery to a given value. You may accomplish this by passing a closure and a value to the `where` method. For example, the following query will retrieve all users who have a recent "membership" of a given type;

```
use App\Models\User;

$users = User::where(function ($query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Or, you may need to construct a "where" clause that compares a column to the results of a subquery. You may accomplish this by passing a column, operator, and closure to the `where` method. For example, the following query will retrieve all income records where the amount is less than average;

```
use App\Models\Income;

$incomes = Income::where('amount', '<', function ($query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Ordering, Grouping, Limit & Offset

Ordering

The `orderBy` Method

The `orderBy` method allows you to sort the results of the query by a given column. The first argument accepted by the `orderBy` method should be the column you wish to sort by, while the second argument determines the

direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

To sort by multiple columns, you may simply invoke `orderBy` as many times as necessary:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->orderBy('email', 'asc')
                ->get();
```

The `latest` & `oldest` Methods

The `latest` and `oldest` methods allow you to easily order results by date. By default, result will be ordered by the table's `created_at` column. Or, you may pass the column name that you wish to sort by:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

Random Ordering

The `inRandomOrder` method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
                ->first();
```

Removing Existing Orderings

The `reorder` method removes all of the "order by" clauses that have previously been applied to the query:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

You may pass a column and direction when calling the `reorder` method in order to remove all existing "order by "clauses" and apply an entirely new order to the query:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

Grouping

The `groupBy` & `having` Methods

As you might expect, the `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

You may pass multiple arguments to the `groupBy` method to group by multiple columns:

```
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

To build more advanced `having` statements, see the `havingRaw` method.

Limit & Offset

The `skip` & `take` Methods

You may use the `skip` and `take` methods to limit the number of results returned from the query or to skip a given number of results in the query:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternatively, you may use the `limit` and `offset` methods. These methods are functionally equivalent to the `take` and `skip` methods, respectively:

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

Conditional Clauses

Sometimes you may want certain query clauses to apply to a query based on another condition. For instance, you may only want to apply a `where` statement if a given input value is present on the incoming HTTP request. You may accomplish this using the `when` method:

```
$role = $request->input('role');

$users = DB::table('users')
                ->when($role, function ($query, $role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

The `when` method only executes the given closure when the first argument is `true`. If the first argument is `false`, the closure will not be executed. So, in the example above, the closure given to the `when` method will only be invoked if the `role` field is present on the incoming request and evaluates to `true`.

You may pass another closure as the third argument to the `when` method. This closure will only execute if the first argument evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default ordering of a query:

```
$sortByVotes = $request->input('sort_by_votes');

$users = DB::table('users')
                ->when($sortByVotes, function ($query, $sortByVotes) {
                    return $query->orderBy('votes');
                }, function ($query) {
                    return $query->orderBy('name');
                })
                ->get();
```

Insert Statements

The query builder also provides an `insert` method that may be used to insert records into the database table. The `insert` method accepts an array of column names and values:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

You may insert several records at once by passing an array of arrays. Each array represents a record that should be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

The `insertOrIgnore` method will ignore duplicate record errors while inserting records into the database:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

> NOTE When using PostgreSQL the `insertGetId` method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the column name as the second parameter to the `insertGetId` method.

Upserts

The `upsert` method will insert records that do not exist and update the records that already exist with new values that you may specify. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of columns that should be updated if a matching record already exists in the database:

```
DB::table('flights')->upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

In the example above, Laravel will attempt to insert two records. If a record already exists with the same `departure` and `destination` column values, Laravel will update that record's `price` column.

> NOTE All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index.

Update Statements

In addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs indicating the columns to be updated. You may constrain the `update` query using `where` clauses:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['votes' => 1]);
```

Update Or Insert

Sometimes you may want to update an existing record in the database or create it if no matching record exists. In this scenario, the `updateOrInsert` method may be used. The `updateOrInsert` method accepts two arguments: an array of conditions by which to find the record, and an array of column and value pairs indicating the columns to be updated.

The `updateOrInsert` method will attempt to locate a matching database record using the first argument's column and value pairs. If the record exists, it will be updated with the values in the second argument. If the record can not be found, a new record will be inserted with the merged attributes of both arguments:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

Updating JSON Columns

When updating a JSON column, you should use `->` syntax to update the appropriate key in the JSON object. This operation is supported on MySQL 5.7+ and PostgreSQL 9.5+:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['options->enabled' => true]);
```

Increment & Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. Both of these methods accept at least one argument: the column to modify. A second argument may be provided to specify the amount by which the column should be incremented or decremented:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

You may also specify additional columns to update during the operation:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Delete Statements

The query builder's `delete` method may be used to delete records from the table. You may constrain `delete` statements by adding "where" clauses before calling the `delete` method:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

If you wish to truncate an entire table, which will remove all records from the table and reset the auto-incrementing ID to zero, you may use the `truncate` method:

```
DB::table('users')->truncate();
```

Table Truncation & PostgreSQL

When truncating a PostgreSQL database, the `CASCADE` behavior will be applied. This means that all foreign key related records in other tables will be deleted as well.

Pessimistic Locking

The query builder also includes a few functions to help you achieve "pessimistic locking" when executing your `select` statements. To execute a statement with a "shared lock", you may call the `sharedLock` method. A shared lock prevents the selected rows from being modified until your transaction is committed:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->sharedLock()
        ->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the selected records from being modified or from being selected with another shared lock:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->lockForUpdate()
        ->get();
```

Debugging

You may use the `dd` and `dump` methods while building a query to dump the current query bindings and SQL. The `dd` method will display the debug information and then stop executing the request. The `dump` method will display the debug information but allow the request to continue executing:

```
DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();
```

Database

Database: Pagination

- Introduction
- Basic Usage
  - Paginating Query Builder Results
  - Paginating Eloquent Results
  - Manually Creating A Paginator
  - Customizing Pagination URLs
- Displaying Pagination Results
  - Adjusting The Pagination Link Window
  - Converting Results To JSON
- Customizing The Pagination View
  - Using Bootstrap
- Paginator Instance Methods

Introduction

In other frameworks, pagination can be very painful. We hope Laravel's approach to pagination will be a breath of fresh air. Laravel's paginator is integrated with the query builder and Eloquent ORM and provides convenient, easy-to-use pagination of database records with zero configuration.

By default, the HTML generated by the paginator is compatible with the Tailwind CSS framework; however, Bootstrap pagination support is also available.

Basic Usage

Paginating Query Builder Results

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent query. The `paginate` method automatically takes care of setting the query's "limit" and "offset" based on the current page being viewed by the user. By default, the current page is detected by the value of the `page` query string argument on the HTTP request. This value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

In this example, the only argument passed to the `paginate` method is the number of items you would like displayed "per page". In this case, let's specify that we would like to display `15` items per page:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show all of the users for the application.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return view('user.index', [
            'users' => DB::table('users')->paginate(15)
        ]);
    }
}
```

"Simple Pagination"

The `paginate` method counts the total number of records matched by the query before retrieving the records from the database. This is done so that the paginator knows how many pages of records there are in total. However, if you do not plan to show the total number of pages in your application's UI then the record count query is unnecessary.

Therefore, if you only need to display simple "Next" and "Previous" links in your application's UI, you may use the `simplePaginate` method to perform a single, efficient query:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries. In this example, we will paginate the `App\Models\User` model and indicate that we plan to display 15 records per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
use App\Models\User;

$users = User::paginate(15);
```

Of course, you may call the `paginate` method after setting other constraints on the query, such as `where` clauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the `simplePaginate` method when paginating Eloquent models:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Manually Creating A Paginator

Sometimes you may wish to create a pagination instance manually, passing it an array of items that you already have in memory. You may do so by creating either an `Illuminate\Pagination\Paginator` or `Illuminate\Pagination\LengthAwarePaginator` instance, depending on your needs.

The `Paginator` class does not need to know the total number of items in the result set; however, because of this, the class does not have methods for retrieving the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`; however, it requires a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simplePaginate` method on the query builder, while the `LengthAwarePaginator` corresponds to the `paginate` method.

> NOTE When manually creating a paginator instance, you should manually "slice" the array of results you pass to the paginator. If you're unsure how to do this, check out the array_slice PHP function.

Customizing Pagination URLs

By default, links generated by the paginator will match the current request's URI. However, the paginator's `withPath` method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like `http://example.com/admin/users?page=N`, you should pass `/admin/users` to the `withPath` method:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->withPath('/admin/users');

    //
});
```

Appending Query String Values

You may append to the query string of pagination links using the `appends` method. For example, to append `sort=votes` to each pagination link, you should make the following call to `appends`:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->appends(['sort' => 'votes']);

    //
});
```

You may use the `withQueryString` method if you would like to append all of the current request's query string values to the pagination links:

```
$users = User::paginate(15)->withQueryString();
```

Appending Hash Fragments

If you need to append a "hash fragment" to URLs generated by the paginator, you may use the `fragment` method. For example, to append `#users` to the end of each pagination link, you should invoke the `fragment` method like so:

```
$users = User::paginate(15)->fragment('users');
```

Displaying Pagination Results

When calling the `paginate` method, you will receive an instance of `Illuminate\Pagination\LengthAwarePaginator`. When calling the `simplePaginate` method, you will receive an instance of `Illuminate\Pagination\Paginator`. These objects provide several methods that describe the result set. In addition to these helpers methods, the paginator instances are iterators and may be looped as an array. So, once you have retrieved the results, you may display the results and render the page links using Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

The `links` method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper `page` query string variable. Remember, the HTML generated by the `links` method is compatible with the Tailwind CSS framework.

Adjusting The Pagination Link Window

When the paginator displays pagination links, the current page number is displayed as well as links for the three pages before and after the current page. If needed, you may control how many additional links are displayed on each side of the current page using the `onEachSide` method:

```
{{ $users->onEachSide(5)->links() }}
```

Converting Results To JSON

The Laravel paginator classes implement the `Illuminate\Contracts\Support\Jsonable` Interface contract and expose the `toJson` method, so it's very easy to convert your pagination results to JSON. You may also convert a paginator instance to JSON by returning it from a route or controller action:

```
use App\Models\User;

Route::get('/users', function () {
    return User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The result records are available via the `data` key in the JSON array. Here is an example of the JSON created by returning a paginator instance from a route:

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "first_page_url": "http://laravel.app?page=1",
   "last_page_url": "http://laravel.app?page=4",
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "path": "http://laravel.app",
   "from": 1,
   "to": 15,
   "data":[
        {
            // Record...
        },
        {
            // Record...
        }
   ]
}
```

Customizing The Pagination View

By default, the views rendered to display the pagination links are compatible with the Tailwind CSS framework. However, if you are not using Tailwind, you are free to define your own views to render these links. When calling the `links` method on a paginator instance, you may pass the view name as the first argument to the method:

```
{{ $paginator->links('view.name') }}

// Passing additional data to the view...
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

However, the easiest way to customize the pagination views is by exporting them to your `resources/views/vendor` directory using the `vendor:publish` command:

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

This command will place the views in your application's `resources/views/vendor/pagination` directory. The `tailwind.blade.php` file within this directory corresponds to the default pagination view. You may edit this file to modify the pagination HTML.

If you would like to designate a different file as the default pagination view, you may invoke the paginator's `defaultView` and `defaultSimpleView` methods within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Paginator::defaultView('view-name');

        Paginator::defaultSimpleView('view-name');
    }
}
```

Using Bootstrap

Laravel includes pagination views built using Bootstrap CSS. To use these views instead of the default Tailwind views, you may call the paginator's `useBootstrap` method within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
use Illuminate\Pagination\Paginator;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Paginator::useBootstrap();
}
```

Paginator Instance Methods

Each paginator instance provides additional pagination information via the following methods:

Method	Description
`$paginator->count()`	Get the number of items for the current page.
`$paginator->currentPage()`	Get the current page number.
`$paginator->firstItem()`	Get the result number of the first item in the results.
`$paginator->getOptions()`	Get the paginator options.
`$paginator->getUrlRange($start, $end)`	Create a range of pagination URLs.
`$paginator->hasPages()`	Determine if there are enough items to split into multiple pages.
`$paginator->hasMorePages()`	Determine if there is more items in the data store.
`$paginator->items()`	Get the items for the current page.
`$paginator->lastItem()`	Get the result number of the last item in the results.
`$paginator->lastPage()`	Get the page number of the last available page. (Not available when using `simplePaginate`).
`$paginator->nextPageUrl()`	Get the URL for the next page.
`$paginator->onFirstPage()`	Determine if the paginator is on the first page.
`$paginator->perPage()`	The number of items to be shown per page.
`$paginator->previousPageUrl()`	Get the URL for the previous page.
`$paginator->total()`	Determine the total number of matching items in the data store. (Not available when using `simplePaginate`).
`$paginator->url($page)`	Get the URL for a given page number.
`$paginator->getPageName()`	Get the query string variable used to store the page.
`$paginator->setPageName($name)`	Set the query string variable used to store the page.

Database

Database: Migrations

- Introduction
- Generating Migrations
  - Squashing Migrations
- Migration Structure
- Running Migrations
  - Rolling Back Migrations
- Tables
  - Creating Tables
  - Updating Tables
  - Renaming / Dropping Tables
- Columns
  - Creating Columns
  - Available Column Types
  - Column Modifiers
  - Modifying Columns
  - Dropping Columns
- Indexes
  - Creating Indexes
  - Renaming Indexes
  - Dropping Indexes
  - Foreign Key Constraints

Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

Generating Migrations

You may use the `make:migration` Artisan command to generate a database migration. The new migration will be placed in your `database/migrations` directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

The `--table` and `--create` options may be used to indicate the name of the table and whether or not the migration will be creating a new table. These options pre-fill the generated migration file with the specified table:

```
php artisan make:migration create_flights_table --create=flights
```

```
php artisan make:migration add_destination_to_flights_table --table=flights
```

If you would like to specify a custom path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The given path should be relative to your application's base path.

> TIP Migration stubs may be customized using stub publishing

Squashing Migrations

As you build your application, you may accumulate more and more migrations over time. This can lead to your `database/migrations` directory becoming bloated with potentially hundreds of migrations. If you would like, you may "squash" your migrations into a single SQL file. To get started, execute the `schema:dump` command:

```
php artisan schema:dump

// Dump the current database schema and prune all existing migrations...
php artisan schema:dump --prune
```

When you execute this command, Laravel will write a "schema" file to your application's `database/schema` directory. Now, when you attempt to migrate your database and no other migrations have been executed, Laravel will execute the schema file's SQL statements first. After executing the schema file's statements, Laravel will execute any remaining migrations that were not part of the schema dump.

You should commit your database schema file to source control so that other new developers on your team may quickly create your application's initial database structure.

> NOTE Migration squashing is only available for the MySQL, PostgreSQL, and SQLite databases and utilizes the database's command-line client. Schema dumps may not be restored to in-memory SQLite databases.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should reverse the operations performed by the `up` method.

Within both of these methods you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, check out its documentation. For example, the following migration creates a `flights` table:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

Forcing Migrations To Run In Production

Some migration operations are destructive, which means they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before the commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To roll back the latest migration operation, you may use the `rollback` Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may roll back a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will roll back the last five migrations:

```
php artisan migrate:rollback --step=5
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Roll Back & Migrate Using A Single Command

The `migrate:refresh` command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

You may roll back and re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will roll back and re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Tables & Migrate

The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

> NOTE The `migrate:fresh` command will drop all database tables regardless of their prefix. This command should be used with caution when developing on a database that is shared with other applications.

Tables

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments: the first is the name of the table, while the second is a closure which receives a `Blueprint` object that may be used to define the new table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
```

```
    $table->timestamps();
});
```

When creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

Database Connection & Table Options

If you want to perform a schema operation on a database connection that is not your application's default connection, use the `connection` method:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

In addition, a few other properties and methods may be used to define other aspects of the table's creation. The `engine` property may be used to specify the table's storage engine when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    // ...
});
```

The `charset` and `collation` properties may be used to specify the character set and collation for the created table when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

The `temporary` method may be used to indicate that the table should be "temporary". Temporary tables are only visible to the current connection's database session and are dropped automatically when the connection is closed:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

Updating Tables

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives a `Blueprint` instance you may use to add columns or indexes to the table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Renaming Tables With Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives an `Illuminate\Database\Schema\Blueprint` instance you may use to add columns to the table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Available Column Types

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

bigIncrements bigInteger binary boolean char dateTimeTz dateTime date decimal double enum float foreignId geometryCollection geometry id increments integer ipAddress json jsonb lineString longText macAddress mediumIncrements mediumInteger mediumText morphs multiLineString multiPoint multiPolygon nullableMorphs nullableTimestamps nullableUuidMorphs point polygon rememberToken set smallIncrements smallInteger softDeletesTz softDeletes string text timeTz time timestampTz timestamp timestampsTz timestamps tinyIncrements tinyInteger unsignedBigInteger unsignedDecimal unsignedInteger unsignedMediumInteger unsignedSmallInteger unsignedTinyInteger uuidMorphs uuid year

`bigIncrements()`

The `bigIncrements` method creates an auto-incrementing `UNSIGNED BIGINT` (primary key) equivalent column:

```
$table->bigIncrements('id');
```

`bigInteger()`

The `bigInteger` method creates a `BIGINT` equivalent column:

```
$table->bigInteger('votes');
```

`binary()`

The `binary` method creates a `BLOB` equivalent column:

```
$table->binary('photo');
```

boolean()

The `boolean` method creates a `BOOLEAN` equivalent column:

```
$table->boolean('confirmed');
```

char()

The `char` method creates a `CHAR` equivalent column with of a given length:

```
$table->char('name', 100);
```

dateTimeTz()

The `dateTimeTz` method creates a `DATETIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->dateTimeTz('created_at', $precision = 0);
```

dateTime()

The `dateTime` method creates a `DATETIME` equivalent column with an optional precision (total digits):

```
$table->dateTime('created_at', $precision = 0);
```

date()

The `date` method creates a `DATE` equivalent column:

```
$table->date('created_at');
```

decimal()

The `decimal` method creates a `DECIMAL` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

double()

The `double` method creates a `DOUBLE` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->double('amount', 8, 2);
```

enum()

The `enum` method creates a `ENUM` equivalent column with the given valid values:

```
$table->enum('difficulty', ['easy', 'hard']);
```

float()

The `float` method creates a `FLOAT` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->float('amount', 8, 2);
```

foreignId()

The `foreignId` method is an alias of the `unsignedBigInteger` method:

```
$table->foreignId('user_id');
```

geometryCollection()

The `geometryCollection` method creates a `GEOMETRYCOLLECTION` equivalent column:

```
$table->geometryCollection('positions');
```

geometry()

The `geometry` method creates a `GEOMETRY` equivalent column:

```
$table->geometry('positions');
```

id()

The `id` method is an alias of the `bigIncrements` method. By default, the method will create an `id` column; however, you may pass a column name if you would like to assign a different name to the column:

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

increments()

The `increments` method creates an auto-incrementing `UNSIGNED INTEGER` equivalent column as a primary key:

```
$table->increments('id');
```

integer()

The `integer` method creates an `INTEGER` equivalent column:

```
$table->integer('votes');
```

ipAddress()

The `ipAddress` method creates an `INTEGER` equivalent column:

```
$table->ipAddress('visitor');
```

json()

The `json` method creates an `JSON` equivalent column:

```
$table->json('options');
```

jsonb()

The `jsonb` method creates an `JSONB` equivalent column:

```
$table->jsonb('options');
```

lineString()

The `lineString` method creates an `LINESTRING` equivalent column:

```
$table->lineString('positions');
```

longText()

The `longText` method creates an `LONGTEXT` equivalent column:

```
$table->longText('description');
```

macAddress()

The `macAddress` method creates a column that is intended to hold a MAC address. Some database systems, such

as PostgreSQL, have a dedicated column type for this type of data. Other database systems will use a string equivalent column:

```
$table->macAddress('device');
```

mediumIncrements()

The `mediumIncrements` method creates an auto-incrementing `UNSIGNED MEDIUMINT` equivalent column as a primary key:

```
$table->mediumIncrements('id');
```

mediumInteger()

The `mediumInteger` method creates an `MEDIUMINT` equivalent column:

```
$table->mediumInteger('votes');
```

mediumText()

The `mediumText` method creates an `MEDIUMTEXT` equivalent column:

```
$table->mediumText('description');
```

morphs()

The `morphs` method is a convenience method that adds a `{column}_id UNSIGNED BIGINT` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->morphs('taggable');
```

multiLineString()

The `multiLineString` method creates an `MULTILINESTRING` equivalent column:

```
$table->multiLineString('positions');
```

multiPoint()

The `multiPoint` method creates an `MULTIPOINT` equivalent column:

```
$table->multiPoint('positions');
```

multiPolygon()

The `multiPolygon` method creates an `MULTIPOLYGON` equivalent column:

```
$table->multiPolygon('positions');
```

nullableTimestamps()

The method is similar to the timestamps method; however, the column that is created will be "nullable":

```
$table->nullableTimestamps(0);
```

nullableMorphs()

The method is similar to the morphs method; however, the columns that are created will be "nullable":

```
$table->nullableMorphs('taggable');
```

nullableUuidMorphs()

The method is similar to the uuidMorphs method; however, the columns that are created will be "nullable":

```
$table->nullableUuidMorphs('taggable');
```

point()

The `point` method creates an `POINT` equivalent column:

```
$table->point('position');
```

polygon()

The `polygon` method creates an `POLYGON` equivalent column:

```
$table->polygon('position');
```

rememberToken()

The `rememberToken` method creates a nullable, `VARCHAR(100)` equivalent column that is intended to store the current "remember me" authentication token:

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

set()

The `set` method creates an `SET` equivalent column with the given list of valid values:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

The `smallIncrements` method creates an auto-incrementing `UNSIGNED SMALLINT` equivalent column as a primary key:

```
$table->smallIncrements('id');
```

smallInteger()

The `smallInteger` method creates an `SMALLINT` equivalent column:

```
$table->smallInteger('votes');
```

softDeletesTz()

The `softDeletesTz` method adds a nullable `deleted_at` `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

softDeletes()

The `softDeletes` method adds a nullable `deleted_at` `TIMESTAMP` equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

string()

The `string` method creates an `VARCHAR` equivalent column of the given length:

```
$table->string('name', 100);
```

text()

The `text` method creates an `TEXT` equivalent column:

```
$table->text('description');
```

timeTz()

The `timeTz` method creates an `TIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timeTz('sunrise', $precision = 0);
```

time()

The `time` method creates an `TIME` equivalent column with an optional precision (total digits):

```
$table->time('sunrise', $precision = 0);
```

timestampTz()

The `timestampTz` method creates an `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timestampTz('added_at', $precision = 0);
```

timestamp()

The `timestamp` method creates an `TIMESTAMP` equivalent column with an optional precision (total digits):

```
$table->timestamp('added_at', $precision = 0);
```

timestampsTz()

The `timestampsTz` method creates `created_at` and `updated_at` `TIMESTAMP` (with timezone) equivalent columns with an optional precision (total digits):

```
$table->timestampsTz($precision = 0);
```

timestamps()

The `timestamps` method creates `created_at` and `updated_at` `TIMESTAMP` equivalent columns with an optional precision (total digits):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

The `tinyIncrements` method creates an auto-incrementing `UNSIGNED TINYINT` equivalent column as a primary key:

```
$table->tinyIncrements('id');
```

tinyInteger()

The `tinyInteger` method creates an `TINYINT` equivalent column:

```
$table->tinyInteger('votes');
```

unsignedBigInteger()

The `unsignedBigInteger` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

The `unsignedDecimal` method creates an `UNSIGNED DECIMAL` equivalent column with an optinoal precision (total digits) and scale (decimal digits):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

`unsignedInteger()`

The `unsignedInteger` method creates an `UNSIGNED INTEGER` equivalent column:

```
$table->unsignedInteger('votes');
```

`unsignedMediumInteger()`

The `unsignedMediumInteger` method creates an `UNSIGNED MEDIUMINT` equivalent column:

```
$table->unsignedMediumInteger('votes');
```

`unsignedSmallInteger()`

The `unsignedSmallInteger` method creates an `UNSIGNED SMALLINT` equivalent column:

```
$table->unsignedSmallInteger('votes');
```

`unsignedTinyInteger()`

The `unsignedTinyInteger` method creates an `UNSIGNED TINYINT` equivalent column:

```
$table->unsignedTinyInteger('votes');
```

`uuidMorphs()`

The `uuidMorphs` method is a convenience method that adds a `{column}_id CHAR(36)` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship that use UUID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->uuidMorphs('taggable');
```

`uuid()`

The `uuid` method creates an `UUID` equivalent column:

```
$table->uuid('id');
```

`year()`

The `year` method creates an `YEAR` equivalent column:

```
$table->year('birth_year');
```

Column Modifiers

In addition to the column types listed above, there are several column "modifiers" you may use when adding a column to a database table. For example, to make the column "nullable", you may use the `nullable` method:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

The following table contains all of the available column modifiers. This list does not include index modifiers:

Modifier	Description
`->after('column')`	Place the column "after" another column (MySQL).
`->autoIncrement()`	Set INTEGER columns as auto-incrementing (primary key).
`->charset('utf8mb4')`	Specify a character set for the column (MySQL).
`->collation('utf8mb4_unicode_ci')`	Specify a collation for the column (MySQL/PostgreSQL/SQL Server).
`->comment('my comment')`	Add a comment to a column (MySQL/PostgreSQL).
`->default($value)`	Specify a "default" value for the column.
`->first()`	Place the column "first" in the table (MySQL).
`->from($integer)`	Set the starting value of an auto-incrementing field (MySQL / PostgreSQL).
`->nullable($value = true)`	Allow NULL values to be inserted into the column.
`->storedAs($expression)`	Create a stored generated column (MySQL).
`->unsigned()`	Set INTEGER columns as UNSIGNED (MySQL).
`->useCurrent()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value.
`->useCurrentOnUpdate()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP when a record is updated.
`->virtualAs($expression)`	Create a virtual generated column (MySQL).
`->generatedAs($expression)`	Create an identity column with specified sequence options (PostgreSQL).
`->always()`	Defines the precedence of sequence values over input for an identity column (PostgreSQL).

Default Expressions

The `default` modifier accepts a value or an `Illuminate\Database\Query\Expression` instance. Using an `Expression` instance will prevent Laravel from wrapping the value in quotes and allow you to use database specific functions. One situation where this is particularly useful is when you need to assign default values to JSON columns:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
}
```

> NOTE Support for default expressions depends on your database driver, database version, and the field type. Please refer to your database's documentation.

Modifying Columns

Prerequisites

Before modifying a column, you must install the `doctrine/dbal` package using the Composer package manager. The Doctrine DBAL library is used to determine the current state of the column and to create the SQL queries needed to make the requested changes to your column:

```
composer require doctrine/dbal
```

If you plan to modify columns created using the `timestamp` method, you must also add the following

configuration to your application's `config/database.php` configuration file:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

Updating Column Attributes

The `change` method allows you to modify the type and attributes of existing columns. For example, you may wish to increase the size of a `string` column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50. To accomplish this, we simply define the new state of the column and then call the `change` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

We could also modify a column to be nullable:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

> NOTE The following column types can be modified: `bigInteger`, `binary`, `boolean`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger`, and `uuid`. To modify a `timestamp` column type a Doctrine type must be registered.

Renaming Columns

To rename a column, you may use the `renameColumn` method provided by the schema builder blueprint. Before renaming a column, ensure that you have installed the `doctrine/dbal` library via the Composer package manager:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

> NOTE Renaming an `enum` column is not currently supported.

Dropping Columns

To drop a column, you may use the `dropColumn` method on the schema builder blueprint. If your application is utilizing an SQLite database, you must install the `doctrine/dbal` package via the Composer package manager before the `dropColumn` method may be used:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

You may drop multiple columns from a table by passing an array of column names to the `dropColumn` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

> NOTE Dropping or modifying multiple columns within a single migration while using a SQLite database is not supported.

Available Command Aliases

Laravel provides several convenient methods related to dropping common types of columns. Each of these

methods is described in the table below:

Command	Description
`$table->dropMorphs('morphable');`	Drop the `morphable_id` and `morphable_type` columns.
`$table->dropRememberToken();`	Drop the `remember_token` column.
`$table->dropSoftDeletes();`	Drop the `deleted_at` column.
`$table->dropSoftDeletesTz();`	Alias of `dropSoftDeletes()` method.
`$table->dropTimestamps();`	Drop the `created_at` and `updated_at` columns.
`$table->dropTimestampsTz();`	Alias of `dropTimestamps()` method.

Indexes

Creating Indexes

The Laravel schema builder supports several types of indexes. The following example creates a new `email` column and specifies that its values should be unique. To create the index, we can chain the `unique` method onto the column definition:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
});
```

Alternatively, you may create the index after defining the column. To do so, you should call the `unique` method on the schema builder blueprint. This methods accepts the name of the column that should receive a unique index:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound (or composite) index:

```
$table->index(['account_id', 'created_at']);
```

When creating an index, Laravel will automatically generate an index name based on the table, column names, and the index type, but you may pass a second argument to the method to specify the index name yourself:

```
$table->unique('email', 'unique_email');
```

Available Index Types

Laravel's schema builder blueprint class provides methods for creating each type of index supported by Laravel. Each index method accepts an optional second argument to specify the name of the index. If omitted, the name will be derived from the names of the table and column(s) used for the index, as well as the index type. Each of the available index methods are described in the table below:

Command	Description
`$table->primary('id');`	Adds a primary key.
`$table->primary(['id', 'parent_id']);`	Adds composite keys.
`$table->unique('email');`	Adds a unique index.
`$table->index('state');`	Adds an index.
`$table->spatialIndex('location');`	Adds a spatial index (except SQLite).

Index Lengths & MySQL / MariaDB

By default, Laravel uses the `utf8mb4` character set. If you are running a version of MySQL older than the 5.7.7 release or MariaDB older than the 10.2.2 release, you may need to manually configure the default string length generated by migrations in order for MySQL to create indexes for them. You may configure the default string length by calling the `Schema::defaultStringLength` method within the `boot` method of your

`App\Providers\AppServiceProvider` class:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Alternatively, you may enable the `innodb_large_prefix` option for your database. Refer to your database's documentation for instructions on how to properly enable this option.

Renaming Indexes

To rename an index, you may use the `renameIndex` method provided by the schema builder blueprint. This method accepts the current index name as its first argument and the desired name as its second argument:

```
$table->renameIndex('from', 'to')
```

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns an index name based on the table name, the name of the indexed column, and the index type. Here are some examples:

Command	Description
`$table->dropPrimary('users_id_primary');`	Drop a primary key from the "users" table.
`$table->dropUnique('users_email_unique');`	Drop a unique index from the "users" table.
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.
`$table->dropSpatialIndex('geo_location_spatialindex');`	Drop a spatial index from the "geo" table (except SQLite).

If you pass an array of columns into a method that drops indexes, the conventional index name will be generated based on the table name, columns, and index type:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

Since this syntax is rather verbose, Laravel provides additional, terser methods that use conventions to provide a better developer experience. The example above can be rewritten like so:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

The `foreignId` method is an alias for `unsignedBigInteger` while the `constrained` method will use conventions to determine the table and column name being referenced. If your table name does not match Laravel's

conventions, you may specify the table name by passing it as an argument to the `constrained` method:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

Any additional column modifiers must be called before the `constrained` method:

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

Dropping Foreign Keys

To drop a foreign key, you may use the `dropForeign` method, passing the name of the foreign key constraint to be deleted as an argument. Foreign key constraints use the same naming convention as indexes. In other words, the foreign key constraint name is based on the name of the table and the columns in the constraint, followed by a "_foreign" suffix:

```
$table->dropForeign('posts_user_id_foreign');
```

Alternatively, you may pass an array containing the column name that holds the foreign key to the `dropForeign` method. The array will be converted to a foreign key constraint name using Laravel's constraint naming conventions:

```
$table->dropForeign(['user_id']);
```

Toggling Foreign Key Constraints

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

> NOTE SQLite disables foreign key constraints by default. When using SQLite, make sure to enable foreign key support in your database configuration before attempting to create them in your migrations. In addition, SQLite only supports foreign keys upon creation of the table and not when tables are altered.

Database

Database: Seeding

- Introduction
- Writing Seeders
  - Using Model Factories
  - Calling Additional Seeders
- Running Seeders

Introduction

Laravel includes the ability to seed your database with test 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.

TIP Mass assignment protection is automatically disabled during database seeding.

Writing Seeders

To generate a seeder, execute the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeders` directory:

```
php artisan make:seeder UserSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the query builder to manually insert data or you may use Eloquent model factories.

As an example, let's modify the default `DatabaseSeeder` class and add a database insert statement to the `run` method:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

TIP You may type-hint any dependencies you need within the `run` method's signature. They will automatically be resolved via the Laravel service container.

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use model factories to conveniently generate large amounts of database records. First, review the model factory

documentation to learn how to define your factories.

For example, let's create 50 users that each have one related post:

```
use App\Models\User;

/**
 * Run the database seeders.
 *
 * @return void
 */
public function run()
{
    User::factory()
            ->times(50)
            ->hasPosts(1)
            ->create();
}
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you may use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes too large. The `call` method accepts an array of seeder classes that should be executed:

```
/**
 * Run the database seeders.
 *
 * @return void
 */
public function run()
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

Running Seeders

You may execute the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `Database\Seeders\DatabaseSeeder` class, which may in turn invoke other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed
```

```
php artisan db:seed --class=UserSeeder
```

You may also seed your database using the `migrate:fresh` command in combination with the `--seed` option, which will drop all tables and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:fresh --seed
```

Forcing Seeders To Run In Production

Some seeding operations may cause you to alter or lose data. In order to protect you from running seeding commands against your production database, you will be prompted for confirmation before the seeders are executed in the `production` environment. To force the seeders to run without a prompt, use the `--force` flag:

```
php artisan db:seed --force
```

Database

Redis

- Introduction
- Configuration
  - Clusters
  - Predis
  - phpredis
- Interacting With Redis
  - Transactions
  - Pipelining Commands
- Pub / Sub

Introduction

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Before using Redis with Laravel, we encourage you to install and use the phpredis PHP extension via PECL. The extension is more complex to install compared to "user-land" PHP packages but may yield better performance for applications that make heavy use of Redis. If you are using Laravel Sail, this extension is already installed in your application's Docker container.

If you are unable to install the phpredis extension, you may install the `predis/predis` package via Composer. Predis is a Redis client written entirely in PHP and does not require any additional extensions:

```
composer require predis/predis
```

Configuration

You may configure your application's Redis settings via the `config/database.php` configuration file. Within this file, you will see a `redis` array containing the Redis servers utilized by your application:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

    'cache' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_CACHE_DB', 1),
    ],

],
```

Each Redis server defined in your configuration file is required to have a name, host, and a port unless you define a single URL to represent the Redis connection:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'url' => 'tcp://127.0.0.1:6379?database=0',
    ],

    'cache' => [
        'url' => 'tls://user:password@127.0.0.1:6380?database=1',
```

```
    ],

],
```

Configuring The Connection Scheme

By default, Redis clients will use the `tcp` scheme when connecting to your Redis servers; however, you may use TLS / SSL encryption by specifying a `scheme` configuration option in your Redis server's configuration array:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'scheme' => 'tls',
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

],
```

Clusters

If your application is utilizing a cluster of Redis servers, you should define these clusters within a `clusters` key of your Redis configuration. This configuration key does not exist by default so you will need to create it within your application's `config/database.php` configuration file:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],

],
```

By default, clusters will perform client-side sharding across your nodes, allowing you to pool nodes and create a large amount of available RAM. However, client-side sharding does not handle failover; therefore, it is primarily suited for transient cached data that is available from another primary data store.

If you would like to use native Redis clustering instead of client-side sharding, you may specify this by setting the `options.cluster` configuration value to `redis` within your application's `config/database.php` configuration file:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],

    'clusters' => [
        // ...
    ],

],
```

Predis

If you would like your application to interact with Redis via the Predis package, you should ensure the

`REDIS_CLIENT` environment variable's value is `predis`:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'predis'),

    // Rest of Redis configuration...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, Predis supports additional connection parameters that may be defined for each of your Redis servers. To utilize these additional configuration options, add them to your Redis server configuration in your application's `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

The Redis Facade Alias

Laravel's `config/app.php` configuration file contains an `aliases` array which defines all of the class aliases that will be registered by the framework. For convenience, an alias entry is included for each facade offered by Laravel; however, the `Redis` alias is disabled because it conflicts with the `Redis` class name provided by the phpredis extension. If you are using the Predis client and would like to enable this alias, you may un-comment the alias in your application's `config/app.php` configuration file.

phpredis

By default, Laravel will use the phpredis extension to communicate with Redis. The client that Laravel will use to communicate with Redis is dictated by the value of the `redis.client` configuration option, which typically reflects the value of the `REDIS_CLIENT` environment variable:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    // Rest of Redis configuration...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, phpredis supports the following additional connection parameters: `name`, `persistent`, `prefix`, `read_timeout`, `retry_interval`, `timeout`, and `context`. You may add any of these options to your Redis server configuration in the `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
    'context' => [
        // 'auth' => ['username', 'secret'],
        // 'stream' => ['verify_peer' => false],
    ],
],
```

Interacting With Redis

You may interact with Redis by calling various methods on the `Redis` facade. The `Redis` facade supports dynamic methods, meaning you may call any Redis command on the facade and the command will be passed directly to Redis. In this example, we will call the Redis `GET` command by calling the `get` method on the `Redis` facade:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Redis;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => Redis::get('user:profile:'.$id)
        ]);
    }
}
```

As mentioned above, you may call any of Redis' commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server. If a Redis command expects arguments, you should pass those to the facade's corresponding method:

```
use Illuminate\Support\Facades\Redis;

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may pass commands to the server using the `Redis` facade's `command` method, which accepts the name of the command as its first argument and an array of values as its second argument:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Using Multiple Redis Connections

Your application's `config/database.php` configuration file allows you to define multiple Redis connections / servers. You may obtain a connection to a specific Redis connection using the `Redis` facade's `connection` method:

```
$redis = Redis::connection('connection-name');
```

To obtain an instance of the default Redis connection, you may call the `connection` method without any additional arguments:

```
$redis = Redis::connection();
```

Transactions

The `Redis` facade's `transaction` method provides a convenient wrapper around Redis' native `MULTI` and `EXEC` commands. The `transaction` method accepts a closure as its only argument. This closure will receive a Redis connection instance and may issue any commands it would like to this instance. All of the Redis commands issued within the closure will be executed in a single, atomic transaction:

```
use Illuminate\Support\Facades\Redis;

Redis::transaction(function ($redis) {
    $redis->incr('user_visits', 1);
    $redis->incr('total_visits', 1);
});
```

> NOTE When defining a Redis transaction, you may not retrieve any values from the Redis connection. Remember, your transaction is executed as a single, atomic operation and that operation is not executed until your entire closure has finished executing its commands.

Lua Scripts

The `eval` method provides another method of executing multiple Redis commands in a single, atomic operation. However, the `eval` method has the benefit of being able to interact with and inspect Redis key values during that operation. Redis scripts are written in the Lua programming language.

The `eval` method can be a bit scary at first, but we'll explore a basic example to break the ice. The `eval` method expects several arguments. First, you should pass the Lua script (as a string) to the method. Secondly, you should pass the number of keys (as an integer) that the script interacts with. Thirdly, you should pass the names of those keys. Finally, you may pass any other additional arguments that you need to access within your script.

In this example, we will increment a counter, inspect its new value, and increment a second counter if the first counter's value is greater than five. Finally, we will return the value of the first counter:

```
$value = Redis::eval(<<<'LUA'
    local counter = redis.call("incr", KEYS[1])

    if counter > 5 then
        redis.call("incr", KEYS[2])
    end

    return counter
LUA, 2, 'first-counter', 'second-counter');
```

NOTE Please consult the Redis documentation for more information on Redis scripting.

Pipelining Commands

Sometimes you may need to execute dozens of Redis commands. Instead of making a network trip to your Redis server for each command, you may use the `pipeline` method. The `pipeline` method accepts one argument: a closure that receives a Redis instance. You may issue all of your commands to this Redis instance and they will all be sent to the Redis server at the same time to reduce network trips to the server. The commands will still be executed in the order they were issued:

```
use Illuminate\Support\Facades\Redis;

Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel provides a convenient interface to the Redis `publish` and `subscribe` commands. These Redis commands allow you to listen for messages on a given "channel". You may publish messages to the channel from another application, or even using another programming language, allowing easy communication between applications and processes.

First, let's setup a channel listener using the `subscribe` method. We'll place this method call within an Artisan command since calling the `subscribe` method begins a long-running process:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
```

```
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function ($message) {
            echo $message;
        });
    }
}
```

Now we may publish messages to the channel using the `publish` method:

```
use Illuminate\Support\Facades\Redis;

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

    Redis::publish('test-channel', json_encode([
        'name' => 'Adam Wathan'
    ]));
});
```

Wildcard Subscriptions

Using the `psubscribe` method, you may subscribe to a wildcard channel, which may be useful for catching all messages on all channels. The channel name will be passed as the second argument to the provided closure:

```
Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {
    echo $message;
});
```

Eloquent ORM

Eloquent: Getting Started

- Introduction
- Generating Model Classes
- Eloquent Model Conventions
  - Table Names
  - Primary Keys
  - Timestamps
  - Database Connections
  - Default Attribute Values
- Retrieving Models
  - Collections
  - Chunking Results
  - Cursors
  - Advanced Subqueries
- Retrieving Single Models / Aggregates
  - Retrieving Or Creating Models
  - Retrieving Aggregates
- Inserting & Updating Models
  - Inserts
  - Updates
  - Mass Assignment
  - Upserts
- Deleting Models
  - Soft Deleting
  - Querying Soft Deleted Models
- Replicating Models
- Query Scopes
  - Global Scopes
  - Local Scopes
- Comparing Models
- Events
  - Using Closures
  - Observers
  - Muting Events

Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

> TIP Before getting started, be sure to configure a database connection in your application's `config/database.php` configuration file. For more information on configuring your database, check out the database configuration documentation.

Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the `app\Models` directory and extend the `Illuminate\Database\Eloquent\Model` class. You may use the `make:model` Artisan command to generate a new model:

```
php artisan make:model Flight
```

If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, and controllers. In addition, these options may be combined to create multiple classes at once:

```
# Generate a model and a FlightFactory class...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generate a model and a FlightSeeder class...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generate a model and a FlightController class...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generate a model and a migration, factory, seeder, and controller...
php artisan make:model Flight -mfsc
```

Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our `Flight` model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `Flight` model stores records in the `flights` table, while an `AirTrafficController` model would store records in an `air_traffic_controllers` table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a `table` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named `id`. If necessary, you may define a protected `$primaryKey` property on your model to specify a different column that serves as your model's primary key:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to `false`:

```
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of `string`:

```
<?php

class Flight extends Model
{
    /**
     * The data type of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

Database Connections

By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The database connection that should be used by the model.
     *
     * @var string
     */
    protected $connection = 'sqlite';
}
```

Default Attribute Values

By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'delayed' => false,
    ];
}
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. The model's `all` method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Building Queries

The Eloquent `all` method will return all of the results in the model's table. However, since each Eloquent model serves as a query builder, you may add additional constraints to queries and then invoke the `get` method to retrieve the results:

```
$flights = Flight::where('active', 1)
               ->orderBy('name')
               ->take(10)
               ->get();
```

> TIP Since Eloquent models are query builders, you should review all of the methods provided by Laravel's query builder. You may use any of these methods when writing your Eloquent queries.

Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

The Eloquent `Collection` class extends Laravel's base `Illuminate\Support\Collection` class, which provides a variety of helpful methods for interacting with data collections. For example, the `reject` method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides a few extra methods that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the `all` or `get` methods. Instead of using these methods, the `chunk` method may be used to process large numbers of models more efficiently.

The `chunk` method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the `chunk` method will provide significantly reduced memory usage when working with a large amount of models:

```
use App\Models\Flight;

Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

The first argument passed to the `chunk` method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the `chunk` method based on a column that you will also be updating while iterating over the results, you should use the `chunkById` method. Using the `chunk` method in these scenarios could lead to unexpected and inconsistent results. Internally, the `chunkById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
        ->chunkById(200, function ($flights) {
            $flights->each->update(['departed' => false]);
        }, $column = 'id');
```

Cursors

Similar to the `chunk` method, the `cursor` method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The `cursor` method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor. Internally, the `cursor` method uses PHP generators to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    //
}
```

The `cursor` returns an `Illuminate\Support\LazyCollection` instance. Lazy collections allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Advanced Subqueries

Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight `destinations` and a table of `flights` to destinations. The `flights` table contains an `arrived_at` column which indicates when the flight arrived at the destination.

Using the subquery functionality available to the query builder's `select` and `addSelect` methods, we can select all of the `destinations` and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

Subquery Ordering

In addition, the query builder's `orderBy` function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to retrieve the first result of a query or perform some other action if no results are found. The `firstOr` method will return the first result matching the query or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the `firstOr` method:

```
$model = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function ($id) {
    return Flight::findOrFail($id);
});
```

Retrieving Or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Retrieving Aggregates

When interacting with Eloquent models, you may also use the `count`, `sum`, `max`, and other aggregate methods provided by the Laravel query builder. As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Inserts

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the `save` method on the model instance:

```
<?php

namespace App\Http\Controllers;

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

```
class FlightController extends Controller
{
    /**
     * Store a new flight in the database.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

In this example, we assign the `name` field from the incoming HTTP request to the `name` attribute of the `App\Models\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The model's `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Alternatively, you may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the `create` method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the mass assignment documentation.

Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated.

> NOTE When issuing a mass update via Eloquent, the `saving`, `saved`, `updating`, and `updated` model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

Examining Attribute Changes

Eloquent provides the `isDirty`, `isClean`, and `wasChanged` methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The `isDirty` method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name to the `isDirty` method to determine if a particular attribute is dirty. The `isClean` will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged('first_name'); // false
```

The `getOriginal` method returns an array containing the original attributes of the model regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

```
$user = User::find(1);

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Array of original attributes...
```

Mass Assignment

You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;

$flight = Flight::create([
```

```
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.

A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Mass Assignment & JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```
/**
 * The attributes that are mass assignable.
 *
 * @var array
 */
$fillable = [
    'options->enabled',
];
```

Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Upserts

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the `save` method.

In the example below, if a flight exists with a `departure` location of `Oakland` and a `destination` location of `San Diego`, it's `price` and `discounted` columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

If you would like to perform multiple "upserts" in a single query, then you should use the `upsert` method instead. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The `upsert` method will automatically set the `created_at` and `updated_at` timestamps if timestamps are enabled on the model:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

> NOTE All databases systems except SQL Server require the columns in the second argument provided to the `upsert` method to have a "primary" or "unique" index.

Deleting Models

To delete a model, you may call the `delete` method on the model instance:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

Deleting An Existing Model By Its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

> NOTE The `destroy` method loads each model individually and calls the `delete` method so that the `deleting` and `deleted` events are properly dispatched for each model.

Deleting Models Using Queries

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deletedRows = Flight::where('active', 0)->delete();
```

NOTE When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

TIP The `SoftDeletes` trait will automatically cast the `deleted_at` attribute to a `DateTime` / `Carbon` instance for you.

You should also add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

```
if ($flight->trashed()) {
    //
}
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the `restore` method on a model instance. The `restore` method will set the model's `deleted_at` column to `null`:

```
$flight->restore();
```

You may also use the `restore` method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

The `restore` method may also be used when building relationship queries:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

```
$flight->forceDelete();
```

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

The `withTrashed` method may also be called when building a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Replicating Models

You may create an unsaved copy of an existing model instance using the `replicate` method. This method is particularly useful when you have model instances that share many of the same attributes:

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

Query Scopes

Global Scopes

Global scopes allow you to add constraints to all queries for a given model. Laravel's own soft delete functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

Writing Global Scopes

Writing a global scope is simple. First, define a class that implements the `Illuminate\Database\Eloquent\Scope` interface. Laravel does not have a conventional location that you should place scope classes, so you are free to place this class in any directory that you wish.

The `Scope` interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints or other types of clauses to the query as needed:

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Apply the scope to a given Eloquent query builder.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $builder
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @return void
     */
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

TIP If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

Applying Global Scopes

To assign a global scope to a model, you should override the model's `booted` method and invoke the model's `addGlobalScope` method. The `addGlobalScope` method accepts an instance of your scope as its only argument:

```
<?php

namespace App\Models;

use App\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

After adding the scope in the example above to the `App\Models\User` model, a call to the `User::all()` method will execute the following SQL query:

```
select * from `users` where `age` > 200
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return a query builder instance:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
```

```
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}
```

Utilizing A Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the `scope` prefix when calling the method. You can even chain calls to various scopes:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of closures to achieve the correct logical grouping:

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" `orWhere` method that allows you to fluently chain scopes together without the use of closures:

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @param  mixed  $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

```
$users = User::ofType('admin')->get();
```

Comparing Models

Sometimes you may need to determine if two models are the "same". The `is` method may be used to quickly verify two models have the same primary key, table, and database connection:

```
if ($post->is($anotherPost)) {
    //
}
```

The `is` method is also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` relationships. This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

```
if ($post->author()->is($user)) {
    //
}
```

Events

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`, and `replicating`.

The `retrieved` event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will dispatch. The `updating` / `updated` events will dispatch when an existing model is modified and the `save` method is called. The `saving` / `saved` events will dispatch when a model is created or updated.

To start listening to model events, define a `$dispatchesEvents` property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own event classes. Each model event class should expect to receive an instance of the affected model via its constructor:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

After defining and mapping your Eloquent events, you may use event listeners to handle the events.

> NOTE When issuing a mass update or delete query via Eloquent, the `saved`, `updated`, `deleting`, and `deleted` model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing a mass updates or deletes.

Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the `booted` method of your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
```

```
    protected static function booted()
    {
        static::created(function ($user) {
            //
        });
    }
}
```

If needed, you may utilize queueable anonymous event listeners when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's queue:

```
use function Illuminate\Events\queueable;

static::created(queueable(function ($user) {
    //
}));
```

Observers

Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as their only argument. The `make:observer` Artisan command is the easiest way to create a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `App/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Handle the User "created" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Handle the User "updated" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function updated(User $user)
    {
        //
    }

    /**
     * Handle the User "deleted" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function deleted(User $user)
    {
        //
    }

    /**
     * Handle the User "forceDeleted" event.
     *
```

```
     * @param  \App\Models\User  $user
     * @return void
     */
    public function forceDeleted(User $user)
    {
        //
    }
}
```

To register an observer, you need to call the `observe` method on the model you wish to observe. You may register observers in the `boot` method of your application's `App\Providers\EventServiceProvider` service provider:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Register any events for your application.
 *
 * @return void
 */
public function boot()
{
    User::observe(UserObserver::class);
}
```

Muting Events

You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events. For example, the following example will fetch and delete an `App\Models\User` instance without dispatching any model events. Any value returned by the closure will be returned by the `withoutEvents` method:

```
use App\Models\User;

$user = User::withoutEvents(function () use () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Saving A Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

Eloquent ORM

Eloquent: Relationships

- Introduction
- Defining Relationships
  - One To One
  - One To Many
  - One To Many (Inverse) / Belongs To
  - Has One Through
  - Has Many Through
- Many To Many Relationships
  - Retrieving Intermediate Table Columns
  - Filtering Queries Via Intermediate Table Columns
  - Defining Custom Intermediate Table Models
- Polymorphic Relationships
  - One To One
  - One To Many
  - Many To Many
  - Custom Polymorphic Types
- Dynamic Relationships
- Querying Relations
  - Relationship Methods Vs. Dynamic Properties
  - Querying Relationship Existence
  - Querying Relationship Absence
  - Querying Morph To Relationships
- Aggregating Related Models
  - Counting Related Models
  - Other Aggregate Functions
  - Counting Related Models On Morph To Relationships
- Eager Loading
  - Constraining Eager Loads
  - Lazy Eager Loading
- Inserting & Updating Related Models
  - The `save` Method
  - The `create` Method
  - Belongs To Relationships
  - Many To Many Relationships
- Touching Parent Timestamps

Introduction

Database tables are often related to one another. For example, a blog post may have many comments or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports a variety of common relationships:

- One To One
- One To Many
- Many To Many
- Has One Through
- Has Many Through
- One To One (Polymorphic)
- One To Many (Polymorphic)
- Many To Many (Polymorphic)

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as

powerful query builders, defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this `posts` relationship:

```
$user->posts()->where('active', 1)->get();
```

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

One To One

A one-to-one relationship is a very basic type of database relationship. For example, a `User` model might be associated with one `Phone` model. To define this relationship, we will place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result. The `hasOne` method is available to your model via the model's `Illuminate\Database\Eloquent\Model` base class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the parent model name. In this case, the `Phone` model is automatically assumed to have a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the primary key column of the parent. In other words, Eloquent will look for the value of the user's `id` column in the `user_id` column of the `Phone` record. If you would like the relationship to use a primary key value other than `id` or your model's `$primaryKey` property, you may pass a third argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User` model. Next, let's define a relationship on the `Phone` model that will let us access the user that owns the phone. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
```

```
    {
        return $this->belongsTo(User::class);
    }
}
```

When invoking the `user` method, Eloquent will attempt to find a `User` model that has an `id` which matches the `user_id` column on the `Phone` model.

Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. So, in this case, Eloquent assumes that the `Phone` model has a `user_id` column. However, if the foreign key on the `Phone` model is not `user_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

If the parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying the parent table's custom key:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

One To Many

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column for the `Comment` model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with `_id`. So, in this example, Eloquent will assume the foreign key column on the `Comment` model is `post_id`.

Once the relationship method has been defined, we can access the collection of related comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were defined as properties on the model:

```
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by

calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = Post::find(1)->comments()
                    ->where('title', 'foo')
                    ->first();
```

Like the `hasOne` method, you may also override the foreign and local keys by passing additional arguments to the `hasMany` method:

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

One To Many (Inverse) / Belongs To

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a `hasMany` relationship, define a relationship method on the child model which calls the `belongsTo` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the `post` "dynamic relationship property":

```
use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;
```

In the example above, Eloquent will attempt to find a `Post` model that has an `id` which matches the `post_id` column on the `Comment` model.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a _ followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the `Post` model's foreign key on the `comments` table is `post_id`.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the `belongsTo` method:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo(Post::class, 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the post that owns the comment.
 */
public function post()
```

```
{
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Default Models

The `belongsTo`, `hasOne`, `hasOneThrough`, and `morphOne` relationships allow you to define a default model that will be returned if the given relationship is `null`. This pattern is often referred to as the Null Object pattern and can help remove conditional checks in your code. In the following example, the `user` relation will return an empty `App\Models\User` model if no user is attached to the `Post` model:

```
/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault();
}
```

To populate the default model with attributes, you may pass an array or closure to the `withDefault` method:

```
/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault(function ($user, $post) {
        $user->name = 'Guest Author';
    });
}
```

Has One Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

For example, in a vehicle repair shop application, each `Mechanic` model may be associated with one `Car` model, and each `Car` model may be associated with one `Owner` model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner *through* the `Car` model. Let's look at the tables necessary to define this relationship:

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Mechanic` model:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

The first argument passed to the `hasOneThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasOneThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars table...
            'car_id', // Foreign key on the owners table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}
```

Has Many Through

The "has-many-through" relationship provides a convenient way to access distant relations via an intermediate relation. For example, let's assume we are building a deployment platform like Laravel Vapor. A `Project` model might access many `Deployment` models through an intermediate `Environment` model. Using this example, you could easily gather all deployments for a given environment. Let's look at the tables required to define this relationship:

```
projects
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Project` model:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;

class Project extends Model
{
    /**
     * Get all of the deployments for the project.
     */
    public function deployments()
    {
        return $this->hasManyThrough(Deployment::class, Environment::class);
    }
}
```

The first argument passed to the `hasManyThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Though the `Deployment` model's table does not contain a `project_id` column, the `hasManyThrough` relation provides access to a project's deployments via `$project->deployments`. To retrieve these models, Eloquent inspects the `project_id` column on the intermediate `Environment` model's table. After finding the relevant environment IDs, they are used to query the `Deployment` model's table.

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Project extends Model
{
    public function deployments()
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Foreign key on the environments table...
            'environment_id', // Foreign key on the deployments table...
            'id', // Local key on the projects table...
            'id' // Local key on the environments table...
        );
    }
}
```

Many To Many Relationships

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

Table Structure

To define this relationship, three database tables are needed: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names and contains `user_id` and `role_id` columns. This table is used as an intermediate table linking the users and roles.

Remember, since a role can belong to many users, we cannot simply place a `user_id` column on the `roles` table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the `role_user` table is needed. We can summarize the relationship's table structure like so:

```
users
    id - integer
    name - string

roles
    id - integer
```

```
    name - string

role_user
    user_id - integer
    role_id - integer
```

Model Structure

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. The `belongsToMany` method is provided by the `Illuminate\Database\Eloquent\Model` base class that is used by all of your application's Eloquent models. For example, let's define a `roles` method on our `User` model. The first argument passed to this method is the name of the related model class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic relationship property:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    //
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `roles` method and continuing to chain conditions onto the query:

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

To determine the table name of the relationship's intermediate table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the `belongsToMany` method:

```
return $this->belongsToMany(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate table, you may also customize the column names of the keys on the table by passing additional arguments to the `belongsToMany` method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the "inverse" of a many-to-many relationship, you should define a method on the related model which also returns the result of the `belongsToMany` method. To complete our user / role example, let's define the `users` method on the `Role` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
```

```
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` model counterpart with the exception of referencing the `App\Models\User` model. Since we're reusing the `belongsToMany` method, all of the usual table and key customization options are available when defining the "inverse" of many-to-many relationships.

Retrieving Intermediate Table Columns

As you have already learned, working with many-to-many relations requires the presence of an intermediate table. Eloquent provides some very helpful ways of interacting with this table. For example, let's assume our `User` model has many `Role` models that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notice that each `Role` model we retrieve is automatically assigned a `pivot` attribute. This attribute contains a model representing the intermediate table.

By default, only the model keys will be present on the `pivot` model. If your intermediate table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

If you would like your intermediate table to have `created_at` and `updated_at` timestamps that are automatically maintained by Eloquent, call the `withTimestamps` method when defining the relationship:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

> NOTE Intermediate tables that utilize Eloquent's automatically maintained timestamps are required to have both `created_at` and `updated_at` timestamp columns.

Customizing The `pivot` Attribute Name

As noted previously, attributes from the intermediate table may be accessed on models via the `pivot` attribute. However, you are free to customize the name of this attribute to better reflect its purpose within your application.

For example, if your application contains users that may subscribe to podcasts, you likely have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table attribute to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany(Podcast::class)
                ->as('subscription')
                ->withTimestamps();
```

Once the custom intermediate table attribute has been specified, you may access the intermediate table data using the customized name:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

Filtering Queries Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` relationship queries using the `wherePivot`, `wherePivotIn`, and `wherePivotNotIn` methods when defining the relationship:

```
return $this->belongsToMany(Role::class)
                ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
                ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
                ->wherePivotNotIn('priority', [1, 2]);
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your many-to-many relationship, you may call the `using` method when defining the relationship. Custom pivot models give you the opportunity to define additional methods on the pivot model.

Custom many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class while custom polymorphic many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\MorphPivot` class. For example, we may define a `Role` model which uses a custom `RoleUser` pivot model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

When defining the `RoleUser` model, you should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    //
}
```

> NOTE Pivot models may not use the `SoftDeletes` trait. If you need to soft delete pivot records consider converting your pivot model to an actual Eloquent model.

Custom Pivot Models And Incrementing IDs

If you have defined a many-to-many relationship that uses a custom pivot model, and that pivot model has an auto-incrementing primary key, you should ensure your custom pivot model class defines an `incrementing` property that is set to `true`.

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

Polymorphic Relationships

A polymorphic relationship allows the child model to belong to more than one type of model using a single association. For example, imagine you are building an application that allows users to share blog posts and videos. a `Comment` model might belong to both the `Post` and `Video` models.

One To One (Polymorphic)

Table Structure

A one-to-one polymorphic relation is similar to a typical one-to-one relation; however, the child model can belong to more than one type of model using a single association. For example, a blog `Post` and a `User` may share a polymorphic relation to an `Image` model. Using a one-to-one polymorphic relation allows you to have a single table of unique images that may be associated with posts and users. First, let's examine the table structure:

```
posts
    id - integer
    name - string

users
    id - integer
    name - string

images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

Note the `imageable_id` and `imageable_type` columns on the `images` table. The `imageable_id` column will contain the ID value of the post or user, while the `imageable_type` column will contain the class name of the parent model. The `imageable_type` column is used by Eloquent to determine which "type" of parent model to return when accessing the `imageable` relation. In this case, the column would contain either `App\Models\Post` or `App\Models\User`.

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
    /**
     * Get the parent imageable model (user or post).
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

class User extends Model
{
```

```
    /**
     * Get the user's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to retrieve the image for a post, we can access the `image` dynamic relationship property:

```
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

You may retrieve the parent of the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `imageable` method on the `Image` model. So, we will access that method as a dynamic relationship property:

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

The `imageable` relation on the `Image` model will return either a `Post` or `User` instance, depending on which type of model owns the image.

Key Conventions

If necessary, you may specify the name of the "id" and "type" columns utilized by your polymorphic child model. If you do so, ensure that you always pass the name of the relationship as the first argument to the `morphTo` method. Typically, this value should match the method name, so you may use PHP's `__FUNCTION__` constant:

```
/**
 * Get the model that the image belongs to.
 */
public function imageable()
{
    return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

One To Many (Polymorphic)

Table Structure

A one-to-many polymorphic relation is similar to a typical one-to-many relation; however, the child model can belong to more than one type of model using a single association. For example, imagine users of your application can "comment" on posts and videos. Using polymorphic relationships, you may use a single `comments` table to contain comments for both posts and videos. First, let's examine the table structure required to build this relationship:

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
```

```
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your model's dynamic relationship properties. For example, to access all of the comments for a post, we can use the `comments` dynamic property:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

You may also retrieve the parent of a polymorphic child model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `commentable` method on the `Comment` model. So, we will access that method as a dynamic relationship property in order to access the comment's parent model:

```
use App\Models\Comment;

$comment = Comment::find(1);

$commentable = $comment->commentable;
```

The `commentable` relation on the `Comment` model will return either a `Post` or `Video` instance, depending on which type of model is the comment's parent.

Many To Many (Polymorphic)

Table Structure

Many-to-many polymorphic relations are slightly more complicated than "morph one" and "morph many" relationships. For example, a `Post` model and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation in this situation would allow your application to have a single table of unique tags that may be associated with posts or videos. First, let's examine the table structure required to build this relationship:

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

> TIP Before diving into polymorphic many-to-many relationships, you may benefit from reading the documentation on typical many-to-many relationships.

Model Structure

Next, we're ready to define the relationships on the models. The `Post` and `Video` models will both contain a `tags` method that calls the `morphToMany` method provided by the base Eloquent model class.

The `morphToMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

Defining The Inverse Of The Relationship

Next, on the `Tag` model, you should define a method for each of its possible parent models. So, in this example, we will define a `posts` method and a `videos` method. Both of these methods should return the result of the `morphedByMany` method.

The `morphedByMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts()
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the tags for a post, you may use the `tags` dynamic relationship property:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

You may retrieve the parent of a polymorphic relation from the polymorphic child model by accessing the name of the method that performs the call to `morphedByMany`. In this case, that is the `posts` or `videos` methods on the `Tag` model:

```
use App\Models\Tag;

$tag = Tag::find(1);

foreach ($tag->posts as $post) {
    //
}

foreach ($tag->videos as $video) {
    //
}
```

Custom Polymorphic Types

By default, Laravel will use the fully qualified class name to store the "type" of the related model. For instance, given the one-to-many relationship example above where a `Comment` model may belong to a `Post` or a `Video` model, the default `commentable_type` would be either `App\Models\Post` or `App\Models\Video`, respectively. However, you may wish to decouple these values from your application's internal structure.

For example, instead of using the model names as the "type", we may use simple strings such as `post` and `video`. By doing so, the polymorphic "type" column values in our database will remain valid even if the models are renamed:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);
```

You may register the `morphMap` in the `boot` function of your `App\Providers\AppServiceProvider` class or create a separate service provider if you wish.

You may determine the morph alias of a given model at runtime using the model's `getMorphClass` method. Conversely, you may determine the fully-qualified class name associated with a morph alias using the `Relation::getMorphedModel` method:

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

> NOTE When adding a "morph map" to your existing application, every morphable `*_type` column value in your database that still contains a fully-qualified class will need to be converted to its "map" name.

Dynamic Relationships

You may use the `resolveRelationUsing` method to define relations between Eloquent models at runtime. While not typically recommended for normal application development, this may occasionally be useful when developing Laravel packages.

The `resolveRelationshipUsing` method accepts the desired relationship name as its first argument. The second argument passed to the method should be a closure that accepts the model instance and returns a valid Eloquent relationship definition. Typically, you should configure dynamic relationships within the boot method of a service provider:

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function ($orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

> NOTE When defining dynamic relationships, always provide explicit key name arguments to the Eloquent relationship methods.

Querying Relations

Since all Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing a query to load the related models. In addition, all types of Eloquent relationships also serve as query builders, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL query against your database.

For example, imagine a blog application in which a `User` model has many associated `Post` models:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

```
use App\Models\User;

$user = User::find(1);

$user->posts()->where('active', 1)->get();
```

You are able to use any of the Laravel query builder's methods on the relationship, so be sure to explore the

query builder documentation to learn about all of the methods that are available to you.

Chaining `orWhere` Clauses After Relationships

As demonstrated in the example above, you are free to add additional constraints to relationships when querying them. However, use caution when chaining `orWhere` clauses onto a relationship, as the `orWhere` clauses will be logically grouped at the same level as the relationship constraint:

```
$user->posts()
        ->where('active', 1)
        ->orWhere('votes', '>=', 100)
        ->get();
```

The example above will generate the following SQL. As you can see, the `or` clause instructs the query to return *any* use with greater than 100 votes. The query is no longer constrained to a specific user:

```
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

In most situations, you should use logical groups to group the conditional checks between parentheses:

```
use Illuminate\Database\Eloquent\Builder;

$user->posts()
        ->where(function (Builder $query) {
            return $query->where('active', 1)
                         ->orWhere('votes', '>=', 100);
        })
        ->get();
```

The example above will produce the following SQL. Note that the logical grouping has properly grouped the constraints and the query remains constrained to a specific user:

```
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may access the relationship as if it were a property. For example, continuing to use our `User` and `Post` example models, we may access all of a user's posts like so:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Dynamic relationship properties perform "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use eager loading to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

Querying Relationship Existence

When retrieving model records, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the `has` and `orHas` methods:

```
use App\Models\Post;

// Retrieve all posts that have at least one comment...
$posts = Post::has('comments')->get();
```

You may also specify an operator and count value to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment that has at least one image:

```
// Retrieve posts that have at least one comment with images...
$posts = Post::has('comments.images')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to define additional query constraints on your `has` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

> NOTE Eloquent does not currently support querying for relationship existence across databases. The relationships must exist within the same database.

Querying Relationship Absence

When retrieving model records, you may wish to limit your results based on the absence of a relationship. For example, imagine you want to retrieve all blog posts that **don't** have any comments. To do so, you may pass the name of the relationship to the `doesntHave` and `orDoesntHave` methods:

```
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to add additional query constraints to your `doesntHave` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

You may use "dot" notation to execute a query against a nested relationship. For example, the following query will retrieve all posts that do not have comments; however, posts that have comments from authors that are not banned will be included in the results:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder $query) {
    $query->where('banned', 0);
})->get();
```

Querying Morph To Relationships

To query the existence of "morph to" relationships, you may use the `whereHasMorph` and `whereDoesntHaveMorph` methods. These methods accept the name of the relationship as their first argument. Next, the methods accept the names of the related models that you wish to include in the query. Finally, you may provide a closure which customizes the relationship query:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;
```

```
// Retrieve comments associated to posts or videos with a title like code%...
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// Retrieve comments associated to posts with a title not like code%...
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();
```

You may occasionally need to add query constraints based on the "type" of the related polymorphic model. The closure passed to the `whereHasMorph` method may receive a `$type` value as its second argument. This argument allows you to inspect the "type" of the query that is being built:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

Querying All Related Models

Instead of passing an array of possible polymorphic models, you may provide `*` as a wildcard value. This will instruct Laravel to retrieve all of the possible polymorphic types from the database. Laravel will execute an additional query in order to perform this operation:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query) {
    $query->where('title', 'like', 'foo%');
})->get();
```

Aggregating Related Models

Counting Related Models

Sometimes you may want to count the number of related models for a given relationship without actually loading the models. To accomplish this, you may use the `withCount` method. The `withCount` method which will place a `{relation}_count` attribute on the resulting models:

```
use App\Models\Post;

$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

By passing an array to the `withCount` method, you may add the "counts" for multiple relations as well as add additional constraints to the queries:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount(['votes', 'comments' => function (Builder $query) {
    $query->where('content', 'like', 'code%');
}])->get();
```

```
echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

You may also alias the relationship count result, allowing multiple counts on the same relationship:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
])->get();

echo $posts[0]->comments_count;
echo $posts[0]->pending_comments_count;
```

Deferred Count Loading

Using the `loadCount` method, you may load a relationship count after the parent model has already been retrieved:

```
$book = Book::first();

$book->loadCount('genres');
```

If you need to set additional query constraints on the count query, you may pass an array keyed by the relationships you wish to count. The array values should be closures which receive the query builder instance:

```
$book->loadCount(['reviews' => function ($query) {
    $query->where('rating', 5);
}])
```

Relationship Counting & Custom Select Statements

If you're combining `withCount` with a `select` statement, ensure that you call `withCount` after the `select` method:

```
$posts = Post::select(['title', 'body'])
                ->withCount('comments')
                ->get();
```

Other Aggregate Functions

In addition to the `withCount` method, Eloquent provides `withMin`, `withMax`, `withAvg`, and `withSum` methods. These methods will place a `{relation}_{function}_{column}` attribute on your resulting models:

```
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

Like the `loadCount` method, deferred versions of these methods are also available. These additional aggregate operations may be performed on Eloquent models that have already been retrieved:

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

Counting Related Models On Morph To Relationships

If you would like to eager load a "morph to" relationship, as well as related model counts for the various entities that may be returned by that relationship, you may utilize the `with` method in combination with the `morphTo` relationship's `morphWithCount` method.

In this example, let's assume that `Photo` and `Post` models may create `ActivityFeed` models. We will assume the `ActivityFeed` model defines a "morph to" relationship named `parentable` that allows us to retrieve the parent

`Photo` or `Post` model for a given `ActivityFeed` instance. Additionally, let's assume that `Photo` models "have many" `Tag` models and `Post` models "have many" `Comment` models.

Now, let's imagine we want to retrieve `ActivityFeed` instances and eager load the `parentable` parent models for each `ActivityFeed` instance. In addition, we want to retrieve the number of tags that are associated with each parent photo and the number of comments that are associated with each parent post:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
    'parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWithCount([
            Photo::class => ['tags'],
            Post::class => ['comments'],
        ]);
    }])->get();
```

Deferred Count Loading

Let's assume we have already retrieved a set of `ActivityFeed` models and now we would like to load the nested relationship counts for the various `parentable` models associated with the activity feeds. You may use the `loadMorphCount` method to accomplish this:

```
$activities = ActivityFeed::with('parentable')->get();

$activities->loadMorphCount('parentable', [
    Photo::class => ['tags'],
    Post::class => ['comments'],
]);
```

Eager Loading

When accessing Eloquent relationships as properties, the related models are "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the "N + 1" query problem. To illustrate the N + 1 query problem, consider a `Book` model that "belongs to" to an `Author` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

Now, let's retrieve all books and their authors:

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute one query to retrieve all of the books within the database table, then another query for each book in order to retrieve the book's author. So, if we have 25 books, the code above would run 26 queries: one for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just two queries. When building a query, you may specify which relationships should be eager loaded using the `with` method:

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed - one query to retrieve all of the books and one query to retrieve all of the authors for all of the books:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships. To do so, just pass an array of relationships to the `with` method:

```
$books = Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager a relationship's relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts:

```
$books = Book::with('author.contacts')->get();
```

Nested Eager Loading `morphTo` Relationships

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `with` method in combination with the `morphTo` relationship's `morphWith` method. To help illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()
    ->with(['parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWith([
            Event::class => ['calendar'],
            Photo::class => ['tags'],
            Post::class => ['author'],
        ]);
    }])->get();
```

Eager Loading Specific Columns

You may not always need every column from the relationships you are retrieving. For this reason, Eloquent

allows you to specify which columns of the relationship you would like to retrieve:

```
$books = Book::with('author:id,name')->get();
```

> NOTE When using this feature, you should always include the `id` column and any relevant foreign key columns in the list of columns you wish to retrieve.

Eager Loading By Default

Sometimes you might want to always load some relationships when retrieving a model. To accomplish this, you may define a `$with` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * The relationships that should always be loaded.
     *
     * @var array
     */
    protected $with = ['author'];

    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

If you would like to remove an item from the `$with` property for a single query, you may use the `without` method:

```
$books = Book::without('author')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship but also specify additional query conditions for the eager loading query. You can accomplish this by passing an array of relationships to the `with` method where the array key is a relationship name and the array value is a closure that adds additional constraints to the eager loading query:

```
use App\Models\User;

$users = User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `code`. You may call other query builder methods to further customize the eager loading operation:

```
$users = User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

> NOTE The `limit` and `take` query builder methods may not be used when constraining eager loads.

Constraining Eager Loading Of `morphTo` Relationships

If you are eager loading a `morphTo` relationship, Eloquent will run multiple queries to fetch each type of related model. You may add additional constraints to each of these queries using the `MorphTo` relation's `constrain` method:

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
    $morphTo->constrain([
        Post::class => function (Builder $query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function (Builder $query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

In this example, Eloquent will only eager load posts that have not been hidden and videos have a `type` value of "educational".

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

```
use App\Models\Book;

$books = Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

If you need to set additional query constraints on the eager loading query, you may pass an array keyed by the relationships you wish to load. The array values should be closure instances which receive the query instance:

```
$author->load(['books' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```
$book->loadMissing('author');
```

Nested Lazy Eager Loading & `morphTo`

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `loadMorph` method.

This method accepts the name of the `morphTo` relationship as its first argument, and an array of model / relationship pairs as its second argument. To help illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
```

Inserting & Updating Related Models

The `save` Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to add a new comment to a post. Instead of manually setting the `post_id` attribute on the `Comment` model you may insert the comment using the relationship's `save` method:

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

Note that we did not access the `comments` relationship as a dynamic property. Instead, we called the `comments` method to obtain an instance of the relationship. The `save` method will automatically add the appropriate `post_id` value to the new `Comment` model.

If you need to save multiple related models, you may use the `saveMany` method:

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
```

The `save` and `saveMany` methods will not add the new models to any in-memory relationships that are already loaded onto the parent model. If you plan on accessing the relationship after using the `save` or `saveMany` methods, you may wish to use the `refresh` method to reload the model and its relationships:

```
$post->comments()->save($comment);

$post->refresh();

// All comments, including the newly saved comment...
$post->comments;
```

Recursively Saving Models & Relationships

If you would like to `save` your model and all of its associated relationships, you may use the `push` method. In this example, the `Post` model will be saved as well as its comments and the comment's authors:

```
$post = Post::find(1);

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

The `create` Method

In addition to the `save` and `saveMany` methods, you may also use the `create` method, which accepts an array of attributes, creates a model, and inserts it into the database. The difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`. The newly created model will be returned by the `create` method:

```
use App\Models\Post;

$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

You may use the `createMany` method to create multiple related models:

```
$post = Post::find(1);

$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

You may also use the `findOrNew`, `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods to create and update models on relationships.

TIP Before using the `create` method, be sure to review the mass assignment documentation.

Belongs To Relationships

If you would like to assign a child model to a new parent model, you may use the `associate` method. In this example, the `User` model defines a `belongsTo` relationship to the `Account` model. This `associate` method will set the foreign key on the child model:

```
use App\Models\Account;

$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

To remove a parent model from a child model, you may use the `dissociate` method. This method will set the relationship's foreign key to `null`:

```
$user->account()->dissociate();

$user->save();
```

Many To Many Relationships

Attaching / Detaching

Eloquent also provides methods to make working with many-to-many relationships more convenient. For example, let's imagine a user can have many roles and a role can have many users. You may use the `attach` method to attach a role to a user by inserting a record in the relationship's intermediate table:

```
use App\Models\User;

$user = User::find(1);

$user->roles()->attach($roleId);
```

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Sometimes it may be necessary to remove a role from a user. To remove a many-to-many relationship record, use the `detach` method. The `detach` method will delete the appropriate record out of the intermediate table; however, both models will remain in the database:

```
// Detach a single role from the user...
$user->roles()->detach($roleId);
```

```
// Detach all roles from the user...
$user->roles()->detach();
```

For convenience, `attach` and `detach` also accept arrays of IDs as input:

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires],
]);
```

Syncing Associations

You may also use the `sync` method to construct many-to-many associations. The `sync` method accepts an array of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the given array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
```

You may also pass additional intermediate table values with the IDs:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

If you do not want to detach existing IDs that are missing from the given array, you may use the `syncWithoutDetaching` method:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

Toggling Associations

The many-to-many relationship also provides a `toggle` method which "toggles" the attachment status of the given related model IDs. If the given ID is currently attached, it will be detached. Likewise, if it is currently detached, it will be attached:

```
$user->roles()->toggle([1, 2, 3]);
```

Updating A Record On The Intermediate Table

If you need to update an existing row in your relationship's intermediate table, you may use the `updateExistingPivot` method. This method accepts the intermediate record foreign key and an array of attributes to update:

```
$user = User::find(1);

$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

Touching Parent Timestamps

When a model defines a `belongsTo` or `belongsToMany` relationship to another model, such as a `Comment` which belongs to a `Post`, it is sometimes helpful to update the parent's timestamp when the child model is updated.

For example, when a `Comment` model is updated, you may want to automatically "touch" the `updated_at` timestamp of the owning `Post` so that it is set to the current date and time. To accomplish this, you may add a `touches` property to your child model containing the names of the relationships that should have their `updated_at` timestamps updated when the child model is updated:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```
class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

NOTE Parent model timestamps will only be updated if the child model is updated using Eloquent's `save` method.

Eloquent ORM

Eloquent: Collections

- Introduction
- Available Methods
- Custom Collections

Introduction

All Eloquent methods that return more than one model result will return instances of the `Illuminate\Database\Eloquent\Collection` class, including results retrieved via the `get` method or accessed via a relationship. The Eloquent collection object extends Laravel's base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models. Be sure to review the Laravel collection documentation to learn all about these helpful methods!

All collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, as previously mentioned, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, we may remove all inactive models and then gather the first name for each remaining user:

```
$names = User::all()->reject(function ($user) {
    return $user->active === false;
})->map(function ($user) {
    return $user->name;
});
```

Eloquent Collection Conversion

While most Eloquent collection methods return a new instance of an Eloquent collection, the `collapse`, `flatten`, `flip`, `keys`, `pluck`, and `zip` methods return a base collection instance. Likewise, if a `map` operation returns a collection that does not contain any Eloquent models, it will be converted to a base collection instance.

Available Methods

All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class.

In addition, the `Illuminate\Database\Eloquent\Collection` class provides a superset of methods to aid with managing your model collections. Most methods return `Illuminate\Database\Eloquent\Collection` instances; however, some methods, like `modelKeys`, return an `Illuminate\Support\Collection` instance.

contains diff except find fresh intersect load loadMissing modelKeys makeVisible makeHidden only toQuery unique

`contains($key, $operator = null, $value = null)`

The `contains` method may be used to determine if a given model instance is contained by the collection. This method accepts a primary key or a model instance:

```
$users->contains(1);

$users->contains(User::find(1));
```

diff($items)

The `diff` method returns all of the models that are not present in the given collection:

```
use App\Models\User;

$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());
```

except($keys)

The `except` method returns all of the models that do not have the given primary keys:

```
$users = $users->except([1, 2, 3]);
```

find($key)

The `find` method returns the model that has a primary key matching the given key. If `$key` is a model instance, `find` will attempt to return a model matching the primary key. If `$key` is an array of keys, `find` will return all models which have a primary key in the given array:

```
$users = User::all();

$user = $users->find(1);
```

fresh($with = [])

The `fresh` method retrieves a fresh instance of each model in the collection from the database. In addition, any specified relationships will be eager loaded:

```
$users = $users->fresh();

$users = $users->fresh('comments');
```

intersect($items)

The `intersect` method returns all of the models that are also present in the given collection:

```
use App\Models\User;

$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());
```

load($relations)

The `load` method eager loads the given relationships for all models in the collection:

```
$users->load(['comments', 'posts']);

$users->load('comments.author');
```

loadMissing($relations)

The `loadMissing` method eager loads the given relationships for all models in the collection if the relationships are not already loaded:

```
$users->loadMissing(['comments', 'posts']);

$users->loadMissing('comments.author');
```

modelKeys()

The `modelKeys` method returns the primary keys for all models in the collection:

```
$users->modelKeys();

// [1, 2, 3, 4, 5]
```

makeVisible($attributes)

The `makeVisible` method makes attributes visible that are typically "hidden" on each model in the collection:

```
$users = $users->makeVisible(['address', 'phone_number']);
```

`makeHidden($attributes)`

The `makeHidden` method hides attributes that are typically "visible" on each model in the collection:

```
$users = $users->makeHidden(['address', 'phone_number']);
```

`only($keys)`

The `only` method returns all of the models that have the given primary keys:

```
$users = $users->only([1, 2, 3]);
```

`toQuery()`

The `toQuery` method returns an Eloquent query builder instance containing a `whereIn` constraint on the collection model's primary keys:

```
use App\Models\User;

$users = User::where('status', 'VIP')->get();

$users->toQuery()->update([
    'status' => 'Administrator',
]);
```

`unique($key = null, $strict = false)`

The `unique` method returns all of the unique models in the collection. Any models of the same type with the same primary key as another model in the collection are removed:

```
$users = $users->unique();
```

Custom Collections

If you would like to use a custom `Collection` object when interacting with a given model, you may define a `newCollection` method on your model:

```
<?php

namespace App\Models;

use App\Support\UserCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Create a new Eloquent Collection instance.
     *
     * @param  array  $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new UserCollection($models);
    }
}
```

Once you have defined a `newCollection` method, you will receive an instance of your custom collection anytime Eloquent would normally return an `Illuminate\Database\Eloquent\Collection` instance. If you would like to use a custom collection for every model in your application, you should define the `newCollection` method on a base model class that is extended by all of your application's models.

Eloquent ORM

Eloquent: Mutators & Casting

- Introduction
- Accessors & Mutators
  - Defining An Accessor
  - Defining A Mutator
- Attribute Casting
  - Array & JSON Casting
  - Date Casting
  - Query Time Casting
- Custom Casts
  - Value Object Casting
  - Array / JSON Serialization
  - Inbound Casting
  - Cast Parameters
  - Castables

Introduction

Accessors, mutators, and attribute casting allows you to transform Eloquent attribute values when you retrieve or set them on model instances. For example, you may want to use the Laravel encrypter to encrypt a value while it is stored in the database, and then automatically decrypt the attribute when you access it on an Eloquent model. Or, you may want to convert a JSON string that is stored in your database to an array when it is accessed via your Eloquent model.

Accessors & Mutators

Defining An Accessor

An accessor transform an Eloquent attribute value when it is accessed. To define an accessor, create a `get{Attribute}Attribute` method on your model where `{Attribute}` is the "studly" cased name of the column you wish to access.

In this example, we'll define an accessor for the `first_name` attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the `first_name` attribute:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the user's first name.
     *
     * @param  string  $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may simply access the `first_name` attribute on a model instance:

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```

You are not limited to interacting with a single attribute within your accessor. You may also use accessors to return new, computed values from existing attributes:

```
/**
 * Get the user's full name.
 *
 * @return string
 */
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}
```

> TIP If you would like these computed values to be added to the array / JSON representations of your model, you will need to append them.

Defining A Mutator

A mutator transforms an Eloquent attribute value when it is set. To define a mutator, define a `set{Attribute}Attribute` method on your model where `{Attribute}` is the "studly" cased name of the column you wish to access.

Let's define a mutator for the `first_name` attribute. This mutator will be automatically called when we attempt to set the value of the `first_name` attribute on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Set the user's first name.
     *
     * @param  string  $value
     * @return void
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

The mutator will receive the value that is being set on the attribute, allowing you to manipulate the value and set the manipulated value on the Eloquent model's internal `$attributes` property. To use our mutator, we only need to set the `first_name` attribute on an Eloquent model:

```
use App\Models\User;

$user = User::find(1);

$user->first_name = 'Sally';
```

In this example, the `setFirstNameAttribute` function will be called with the value `Sally`. The mutator will then apply the `strtolower` function to the name and set its resulting value in the internal `$attributes` array.

Attribute Casting

Attribute casting provides functionality similar to accessors and mutators without requiring you to define any additional methods on your model. Instead, your model's `$casts` property provides a convenient method of converting attributes to common data types.

The `$casts` property should be an array where the key is the name of the attribute being cast and the value is the type you wish to cast the column to. The supported cast types are:

- `array`
- `boolean`
- `collection`
- `date`
- `datetime`
- `decimal:<digits>`
- `double`
- `encrypted`
- `encrypted:array`
- `encrypted:collection`
- `encrypted:object`
- `float`
- `integer`
- `object`
- `real`
- `string`
- `timestamp`

To demonstrate attribute casting, let's cast the `is_admin` attribute, which is stored in our database as an integer (`0` or `1`) to a boolean value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

After defining the cast, the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer:

```
$user = App\Models\User::find(1);

if ($user->is_admin) {
    //
}
```

> NOTE Attributes that are `null` will not be cast. In addition, you should never define a cast (or an attribute) that has the same name as a relationship.

Array & JSON Casting

The `array` cast is particularly useful when working with columns that are stored as serialized JSON. For example, if your database has a `JSON` or `TEXT` field type that contains serialized JSON, adding the `array` cast to that attribute will automatically deserialize the attribute to a PHP array when you access it on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
```

```
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

Once the cast is defined, you may access the `options` attribute and it will automatically be deserialized from JSON into a PHP array. When you set the value of the `options` attribute, the given array will automatically be serialized back into JSON for storage:

```
use App\Models\User;

$user = User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

To update a single field of a JSON attribute with a more terse syntax, you may use the `->` operator when calling the `update` method:

```
$user = User::find(1);

$user->update(['options->key' => 'value']);
```

Date Casting

By default, Eloquent will cast the `created_at` and `updated_at` columns to instances of Carbon, which extends the PHP `DateTime` class and provides an assortment of helpful methods. You may cast additional date attributes by defining additional date casts within your model's `$cast` property array. Typically, dates should be cast using the `datetime` cast.

When defining a `date` or `datetime` cast, you may also specify the date's format. This format will be used when the model is serialized to an array or JSON:

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'created_at' => 'datetime:Y-m-d',
];
```

When a column is cast as a date, you may set its value to a UNIX timestamp, date string (`Y-m-d`), date-time string, or a `DateTime` / `Carbon` instance. The date's value will be correctly converted and stored in your database:

You may customize the default serialization format for all of your model's dates by defining a `serializeDate` method on your model. This method does not affect how your dates are formatted for storage in the database:

```
/**
 * Prepare a date for array / JSON serialization.
 *
 * @param  \DateTimeInterface  $date
 * @return string
 */
protected function serializeDate(DateTimeInterface $date)
{
    return $date->format('Y-m-d');
}
```

To specify the format that should be used when actually storing a model's dates within your database, you should define a `$dateFormat` property on your model:

```
/**
 * The storage format of the model's date columns.
 *
 * @var string
 */
protected $dateFormat = 'U';
```

Query Time Casting

Sometimes you may need to apply casts while executing a query, such as when selecting a raw value from a table. For example, consider the following query:

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

$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->get();
```

The `last_posted_at` attribute on the results of this query will be a simple string. It would be wonderful if we could apply a `datetime` cast to this attribute when executing the query. Thankfully, we may accomplish this using the `withCasts` method:

```
$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->withCasts([
    'last_posted_at' => 'datetime'
])->get();
```

Custom Casts

Laravel has a variety of built-in, helpful cast types; however, you may occasionally need to define your own cast types. You may accomplish this by defining a class that implements the `CastsAttributes` interface.

Classes that implement this interface must define a `get` and `set` method. The `get` method is responsible for transforming a raw value from the database into a cast value, while the `set` method should transform a cast value into a raw value that can be stored in the database. As an example, we will re-implement the built-in `json` cast type as a custom cast type:

```
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class Json implements CastsAttributes
{
    /**
     * Cast the given value.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  mixed  $value
     * @param  array  $attributes
     * @return array
     */
    public function get($model, $key, $value, $attributes)
    {
        return json_decode($value, true);
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  array  $value
     * @param  array  $attributes
     * @return string
```

```
     */
    public function set($model, $key, $value, $attributes)
    {
        return json_encode($value);
    }
}
```

Once you have defined a custom cast type, you may attach it to a model attribute using its class name:

```
<?php

namespace App\Models;

use App\Casts\Json;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => Json::class,
    ];
}
```

Value Object Casting

You are not limited to casting values to primitive types. You may also cast values to objects. Defining custom casts that cast values to objects is very similar to casting to primitive types; however, the `set` method should return an array of key / value pairs that will be used to set raw, storable values on the model.

As an example, we will define a custom cast class that casts multiple model values into a single `Address` value object. We will assume the `Address` value has two public properties: `lineOne` and `lineTwo`:

```
<?php

namespace App\Casts;

use App\Models\Address as AddressModel;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use InvalidArgumentException;

class Address implements CastsAttributes
{
    /**
     * Cast the given value.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  mixed  $value
     * @param  array  $attributes
     * @return \App\Models\Address
     */
    public function get($model, $key, $value, $attributes)
    {
        return new AddressModel(
            $attributes['address_line_one'],
            $attributes['address_line_two']
        );
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  \App\Models\Address  $value
     * @param  array  $attributes
     * @return array
     */
    public function set($model, $key, $value, $attributes)
    {
        if (! $value instanceof AddressModel) {
```

```
            throw new InvalidArgumentException('The given value is not an Address instance.');
        }

        return [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ];
    }
}
```

When casting to value objects, any changes made to the value object will automatically be synced back to the model before the model is saved:

```
use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Value';

$user->save();
```

> TIP If you plan to serialize your Eloquent models containing value objects to JSON or arrays, you should implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces on the value object.

Array / JSON Serialization

When an Eloquent model is converted to an array or JSON using the `toArray` and `toJson` methods, your custom cast value objects will typically be serialized as well as long as they implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces. However, when using value objects provided by third-party libraries, you may not have the ability to add these interfaces to the object.

Therefore, you may specify that your custom cast class will be responsible for serializing the value object. To do so, your custom class cast should implement the `Illuminate\Contracts\Database\Eloquent\SerializesCastableAttributes` interface. This interface states that your class should contain a `serialize` method which should return the serialized form of your value object:

```
/**
 * Get the serialized representation of the value.
 *
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @param  string  $key
 * @param  mixed  $value
 * @param  array  $attributes
 * @return mixed
 */
public function serialize($model, string $key, $value, array $attributes)
{
    return (string) $value;
}
```

Inbound Casting

Occasionally, you may need to write a custom cast that only transforms values that are being set on the model and does not perform any operations when attributes are being retrieved from the model. A classic example of an inbound only cast is a "hashing" cast. Inbound only custom casts should implement the `CastsInboundAttributes` interface, which only requires a `set` method to be defined.

```
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsInboundAttributes;

class Hash implements CastsInboundAttributes
{
    /**
     * The hashing algorithm.
     *
     * @var string
     */
```

```
    protected $algorithm;

    /**
     * Create a new cast class instance.
     *
     * @param  string|null  $algorithm
     * @return void
     */
    public function __construct($algorithm = null)
    {
        $this->algorithm = $algorithm;
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  array  $value
     * @param  array  $attributes
     * @return string
     */
    public function set($model, $key, $value, $attributes)
    {
        return is_null($this->algorithm)
                    ? bcrypt($value)
                    : hash($this->algorithm, $value);
    }
}
```

Cast Parameters

When attaching a custom cast to a model, cast parameters may be specified by separating them from the class name using a `:` character and comma-delimiting multiple parameters. The parameters will be passed to the constructor of the cast class:

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'secret' => Hash::class.':sha256',
];
```

Castables

You may want to allow your application's value objects to define their own custom cast classes. Instead of attaching the custom cast class to your model, you may alternatively attach a value object class that implements the `Illuminate\Contracts\Database\Eloquent\Castable` interface:

```
use App\Models\Address;

protected $casts = [
    'address' => Address::class,
];
```

Objects that implement the `Castable` interface must define a `castUsing` method that returns the class name of the custom caster class that is responsible for casting to and from the `Castable` class:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use App\Casts\Address as AddressCast;

class Address implements Castable
{
    /**
     * Get the name of the caster class to use when casting from / to this cast target.
     *
     * @param  array  $arguments
     * @return string
```

```
     */
    public static function castUsing(array $arguments)
    {
        return AddressCast::class;
    }
}
```

When using `Castable` classes, you may still provide arguments in the `$casts` definition. The arguments will be passed to the `castUsing` method:

```
use App\Models\Address;

protected $casts = [
    'address' => Address::class.':argument',
];
```

Castables & Anonymous Cast Classes

By combining "castables" with PHP's anonymous classes, you may define a value object and its casting logic as a single castable object. To accomplish this, return an anonymous class from your value object's `castUsing` method. The anonymous class should implement the `CastsAttributes` interface:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class Address implements Castable
{
    // ...

    /**
     * Get the caster class to use when casting from / to this cast target.
     *
     * @param  array  $arguments
     * @return object|string
     */
    public static function castUsing(array $arguments)
    {
        return new class implements CastsAttributes
        {
            public function get($model, $key, $value, $attributes)
            {
                return new Address(
                    $attributes['address_line_one'],
                    $attributes['address_line_two']
                );
            }

            public function set($model, $key, $value, $attributes)
            {
                return [
                    'address_line_one' => $value->lineOne,
                    'address_line_two' => $value->lineTwo,
                ];
            }
        };
    }
}
```

Eloquent ORM

Eloquent: API Resources

- Introduction
- Generating Resources
- Concept Overview
  - Resource Collections
- Writing Resources
  - Data Wrapping
  - Pagination
  - Conditional Attributes
  - Conditional Relationships
  - Adding Meta Data
- Resource Responses

Introduction

When building an API, you may need a transformation layer that sits between your Eloquent models and the JSON responses that are actually returned to your application's users. For example, you may wish to display certain attributes for a subset of users and not others, or you may wish to always include certain relationships in the JSON representation of your models. Eloquent's resource classes allow you to expressively and easily transform your models and model collections into JSON.

Of course, you may always convert Eloquent models or collections to JSON using their `toJson` methods; however, Eloquent resources provide more granular and robust control over the JSON serialization of your models and their relationships.

Generating Resources

To generate a resource class, you may use the `make:resource` Artisan command. By default, resources will be placed in the `app/Http/Resources` directory of your application. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource User
```

Resource Collections

In addition to generating resources that transform individual models, you may generate resources that are responsible for transforming collections of models. This allows your JSON responses to include links and other meta information that is relevant to an entire collection of a given resource.

To create a resource collection, you should use the `--collection` flag when creating the resource. Or, including the word `Collection` in the resource name will indicate to Laravel that it should create a collection resource. Collection resources extend the `Illuminate\Http\Resources\Json\ResourceCollection` class:

```
php artisan make:resource User --collection
```

```
php artisan make:resource UserCollection
```

Concept Overview

> TIP This is a high-level overview of resources and resource collections. You are highly encouraged to read the other sections of this documentation to gain a deeper understanding of the customization and power offered to you by resources.

Before diving into all of the options available to you when writing resources, let's first take a high-level look at how resources are used within Laravel. A resource class represents a single model that needs to be transformed into a JSON structure. For example, here is a simple `User` resource class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Every resource class defines a `toArray` method which returns the array of attributes that should be converted to JSON when the resource is returned as a response from a route or controller method.

Note that we can access model properties directly from the `$this` variable. This is because a resource class will automatically proxy property and method access down to the underlying model for convenient access. Once the resource is defined, it may be returned from a route or controller. The resource accepts the underlying model instance via its constructor:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

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

Resource Collections

If you are returning a collection of resources or a paginated response, you should use the `collection` method provided by your resource class when creating the resource instance in your route or controller:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Note that this does not allow any addition of custom meta data that may need to be returned with your collection. If you would like to customize the resource collection response, you may create a dedicated resource to represent the collection:

```
php artisan make:resource UserCollection
```

Once the resource collection class has been generated, you may easily define any meta data that should be included with the response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
```

```
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

After defining your resource collection, it may be returned from a route or controller:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Preserving Collection Keys

When returning a resource collection from a route, Laravel resets the collection's keys so that they are in numerical order. However, you may add a `preserveKeys` property to your resource class indicating whether a collection's original keys should be preserved:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

When the `preserveKeys` property is set to `true`, collection keys will be preserved when the collection is returned from a route or controller:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

Customizing The Underlying Resource Class

Typically, the `$this->collection` property of a resource collection is automatically populated with the result of mapping each item of the collection to its singular resource class. The singular resource class is assumed to be the collection's class name without the trailing `Collection` portion of the class name.

For example, `UserCollection` will attempt to map the given user instances into the `User` resource. To customize this behavior, you may override the `$collects` property of your resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * The resource that this resource collects.
```

```
     *
     * @var string
     */
    public $collects = Member::class;
}
```

Writing Resources

TIP If you have not read the concept overview, you are highly encouraged to do so before proceeding with this documentation.

In essence, resources are simple. They only need to transform a given model into an array. So, each resource contains a `toArray` method which translates your model's attributes into an API friendly array that can be returned from your application's routes or controllers:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Once a resource has been defined, it may be returned directly from a route or controller:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

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

Relationships

If you would like to include related resources in your response, you may add them to the array returned by your resource's `toArray` method. In this example, we will use the `Post` resource's `collection` method to add the user's blog posts to the resource response:

```
use App\Http\Resources\Post;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => Post::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
```

```
}
```

TIP If you would like to include relationships only when they have already been loaded, check out the documentation on conditional relationships.

Resource Collections

While resources transform a single model into an array, resource collections transform a collection of models into an array. However, it is not absolutely necessary to define a resource collection class for each one of your models since all resources provide a `collection` method to generate an "ad-hoc" resource collection on the fly:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

However, if you need to customize the meta data returned with the collection, it is necessary to define your own resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Like singular resources, resource collections may be returned directly from routes or controllers:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Data Wrapping

By default, your outermost resource is wrapped in a `data` key when the resource response is converted to JSON. So, for example, a typical resource collection response looks like the following:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ]
```

```
}
```

If you would like to use a custom key instead of `data`, you may define a `$wrap` attribute on the resource class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * The "data" wrapper that should be applied.
     *
     * @var string
     */
    public static $wrap = 'user';
}
```

If you would like to disable the wrapping of the outermost resource, you should invoke the `withoutWrapping` method on the base `Illuminate\Http\Resources\Json\JsonResource` class. Typically, you should call this method from your `AppServiceProvider` or another service provider that is loaded on every request to your application:

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        JsonResource::withoutWrapping();
    }
}
```

> NOTE The `withoutWrapping` method only affects the outermost response and will not remove `data` keys that you manually add to your own resource collections.

Wrapping Nested Resources

You have total freedom to determine how your resource's relationships are wrapped. If you would like all resource collections to be wrapped in a `data` key, regardless of their nesting, you should define a resource collection class for each resource and return the collection within a `data` key.

You may be wondering if this will cause your outermost resource to be wrapped in two `data` keys. Don't worry, Laravel will never let your resources be accidentally double-wrapped, so you don't have to be concerned about the nesting level of the resource collection you are transforming:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
```

```
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

Data Wrapping And Pagination

When returning paginated collections via a resource response, Laravel will wrap your resource data in a `data` key even if the `withoutWrapping` method has been called. This is because paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Pagination

You may pass a Laravel paginator instance to the `collection` method of a resource or to a custom resource collection:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
```

```
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Conditional Attributes

Sometimes you may wish to only include an attribute in a resource response if a given condition is met. For example, you may wish to only include a value if the current user is an "administrator". Laravel provides a variety of helper methods to assist you in this situation. The `when` method may be used to conditionally add an attribute to a resource response:

```
use Illuminate\Support\Facades\Auth;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when(Auth::user()->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, the `secret` key will only be returned in the final resource response if the authenticated user's `isAdmin` method returns `true`. If the method returns `false`, the `secret` key will be removed from the resource response before it is sent to the client. The `when` method allows you to expressively define your resources without resorting to conditional statements when building the array.

The `when` method also accepts a closure as its second argument, allowing you to calculate the resulting value only if the given condition is `true`:

```
'secret' => $this->when(Auth::user()->isAdmin(), function () {
    return 'secret-value';
}),
```

Merging Conditional Attributes

Sometimes you may have several attributes that should only be included in the resource response based on the same condition. In this case, you may use the `mergeWhen` method to include the attributes in the response only when the given condition is `true`:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
```

```
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen(Auth::user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

Again, if the given condition is `false`, these attributes will be removed from the resource response before it is sent to the client.

> NOTE The `mergeWhen` method should not be used within arrays that mix string and numeric keys. Furthermore, it should not be used within arrays with numeric keys that are not ordered sequentially.

Conditional Relationships

In addition to conditionally loading attributes, you may conditionally include relationships on your resource responses based on if the relationship has already been loaded on the model. This allows your controller to decide which relationships should be loaded on the model and your resource can easily include them only when they have actually been loaded. Ultimately, this makes it easier to avoid "N+1" query problems within your resources.

The `whenLoaded` method may be used to conditionally load a relationship. In order to avoid unnecessarily loading relationships, this method accepts the name of the relationship instead of the relationship itself:

```
use App\Http\Resources\Post;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => Post::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, if the relationship has not been loaded, the `posts` key will be removed from the resource response before it is sent to the client.

Conditional Pivot Information

In addition to conditionally including relationship information in your resource responses, you may conditionally include data from the intermediate tables of many-to-many relationships using the `whenPivotLoaded` method. The `whenPivotLoaded` method accepts the name of the pivot table as its first argument. The second argument should be a closure that returns the value to be returned if the pivot information is available on the model:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_user', function () {
```

```
            return $this->pivot->expires_at;
        }),
    ];
}
```

If your relationship is using a custom intermediate table model, you may pass an instance of the intermediate table model as the first argument to the `whenPivotLoaded` method:

```
'expires_at' => $this->whenPivotLoaded(new Membership, function () {
    return $this->pivot->expires_at;
}),
```

If your intermediate table is using an accessor other than `pivot`, you may use the `whenPivotLoadedAs` method:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function () {
            return $this->subscription->expires_at;
        }),
    ];
}
```

Adding Meta Data

Some JSON API standards require the addition of meta data to your resource and resource collections responses. This often includes things like `links` to the resource or related resources, or meta data about the resource itself. If you need to return additional meta data about a resource, include it in your `toArray` method. For example, you might include `link` information when transforming a resource collection:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

When returning additional meta data from your resources, you never have to worry about accidentally overriding the `links` or `meta` keys that are automatically added by Laravel when returning paginated responses. Any additional `links` you define will be merged with the links provided by the paginator.

Top Level Meta Data

Sometimes you may wish to only include certain meta data with a resource response if the resource is the outermost resource being returned. Typically, this includes meta information about the response as a whole. To define this meta data, add a `with` method to your resource class. This method should return an array of meta data to be included with the resource response only when the resource is the outermost resource being transformed:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;
```

```
class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function with($request)
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}
```

Adding Meta Data When Constructing Resources

You may also add top-level data when constructing resource instances in your route or controller. The `additional` method, which is available on all resources, accepts an array of data that should be added to the resource response:

```
return (new UserCollection(User::all()->load('roles')))
                ->additional(['meta' => [
                    'key' => 'value',
                ]]);
```

Resource Responses

As you have already read, resources may be returned directly from routes and controllers:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

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

However, sometimes you may need to customize the outgoing HTTP response before it is sent to the client. There are two ways to accomplish this. First, you may chain the `response` method onto the resource. This method will return an `Illuminate\Http\JsonResponse` instance, giving you full control over the response's headers:

```
use App\Http\Resources\User as UserResource;
use App\Models\User;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
                ->response()
                ->header('X-Value', 'True');
});
```

Alternatively, you may define a `withResponse` method within the resource itself. This method will be called when the resource is returned as the outermost resource in a response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;
```

```
class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Illuminate\Http\Response  $response
     * @return void
     */
    public function withResponse($request, $response)
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent ORM

Eloquent: Serialization

- Introduction
- Serializing Models & Collections
  - Serializing To Arrays
  - Serializing To JSON
- Hiding Attributes From JSON
- Appending Values To JSON
- Date Serialization

Introduction

When building APIs using Laravel, you will often need to convert your models and relationships to arrays or JSON. Eloquent includes convenient methods for making these conversions, as well as controlling which attributes are included in the serialized representation of your models.

> TIP For an even more robust way of handling Eloquent model and collection JSON serialization, check out the documentation on Eloquent API resources.

Serializing Models & Collections

Serializing To Arrays

To convert a model and its loaded relationships to an array, you should use the `toArray` method. This method is recursive, so all attributes and all relations (including the relations of relations) will be converted to arrays:

```
use App\Models\User;

$user = User::with('roles')->first();

return $user->toArray();
```

The `attributesToArray` method may be used to convert a model's attributes to an array but not its relationships:

```
$user = User::first();

return $user->attributesToArray();
```

You may also convert entire collections of models to arrays by calling the `toArray` method on the collection instance:

```
$users = User::all();

return $users->toArray();
```

Serializing To JSON

To convert a model to JSON, you should use the `toJson` method. Like `toArray`, the `toJson` method is recursive, so all attributes and relations will be converted to JSON. You may also specify any JSON encoding options that are supported by PHP:

```
use App\Models\User;

$user = User::find(1);

return $user->toJson();

return $user->toJson(JSON_PRETTY_PRINT);
```

Alternatively, you may cast a model or collection to a string, which will automatically call the `toJson` method on the model or collection:

```
return (string) User::find(1);
```

Since models and collections are converted to JSON when cast to a string, you can return Eloquent objects directly from your application's routes or controllers. Laravel will automatically serialize your Eloquent models and collections to JSON when they are returned from routes or controllers:

```
Route::get('users', function () {
    return User::all();
});
```

Relationships

When an Eloquent model is converted to JSON, its loaded relationships will automatically be included as attributes on the JSON object. Also, though Eloquent relationship methods are defined using "camel case" method names, a relationship's JSON attribute will be "snake case".

Hiding Attributes From JSON

Sometimes you may wish to limit the attributes, such as passwords, that are included in your model's array or JSON representation. To do so, add a `$hidden` property to your model. In attributes that are listed in the `$hidden` property's array will not be included in the serialized representation of your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

TIP To hide relationships, add the relationship's method name to your Eloquent model's `$hidden` property.

Alternatively, you may use the `visible` property to define an "allow list" of attributes that should be included in your model's array and JSON representation. All attributes that are not present in the `$visible` array will be hidden when the model is converted to an array or JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be visible in arrays.
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

Temporarily Modifying Attribute Visibility

If you would like to make some typically hidden attributes visible on a given model instance, you may use the `makeVisible` method. The `makeVisible` method returns the model instance:

```
return $user->makeVisible('attribute')->toArray();
```

Likewise, if you would like to hide some attributes that are typically visible, you may use the `makeHidden` method.

```
return $user->makeHidden('attribute')->toArray();
```

Appending Values To JSON

Occasionally, when converting models to arrays or JSON, you may wish to add attributes that do not have a corresponding column in your database. To do so, first define an accessor for the value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Determine if the user is an administrator.
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] === 'yes';
    }
}
```

After creating the accessor, add the attribute name to the `appends` property of your model. Note that attribute names are typically referenced using their "snake case" serialized representation, even though the accessor's PHP method is defined using "camel case":

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The accessors to append to the model's array form.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

Once the attribute has been added to the `appends` list, it will be included in both the model's array and JSON representations. Attributes in the `appends` array will also respect the `visible` and `hidden` settings configured on the model.

Appending At Run Time

At runtime, you may instruct a model instance to append additional attributes using the `append` method. Or, you may use the `setAppends` method to override the entire array of appended properties for a given model instance:

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

Date Serialization

Customizing The Default Date Format

You may customize the default serialization format by overriding the `serializeDate` method. This method does not affect how your dates are formatted for storage in the database:

```
/**
 * Prepare a date for array / JSON serialization.
 *
 * @param  \DateTimeInterface  $date
```

```
 * @return string
 */
protected function serializeDate(DateTimeInterface $date)
{
    return $date->format('Y-m-d');
}
```

Customizing The Date Format Per Attribute

You may customize the serialization format of individual Eloquent date attributes by specifying the date format in the model's cast declarations:

```
protected $casts = [
    'birthday' => 'date:Y-m-d',
    'joined_at' => 'datetime:Y-m-d H:00',
];
```

Testing

Testing: Getting Started

- Introduction
- Environment
- Creating & Running Tests
  - Artisan Test Runner

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a `phpunit.xml` file is already set up for your application. The framework also ships with convenient helper methods that allow you to expressively test your applications.

By default, your application's `tests` directory contains two directories: `Feature` and `Unit`. Unit tests are tests that focus on a very small, isolated portion of your code. In fact, most unit tests probably focus on a single method. Tests within your "Unit" test directory do not boot your Laravel application and therefore are unable to access your application's database or other framework services.

Feature tests may test a larger portion of your code, including how several objects interact with each other or even a full HTTP request to a JSON endpoint. **Generally, most of your tests should be feature tests. These types of tests provide the most confidence that your system as a whole is functioning as intended.**

An `ExampleTest.php` file is provided in both the `Feature` and `Unit` test directories. After installing a new Laravel application, execute the `vendor/bin/phpunit` or `php artisan test` commands to run your tests.

Environment

When running tests via `vendor/bin/phpunit`, Laravel will automatically set the configuration environment to `testing` because of the environment variables defined in the `phpunit.xml` file. Laravel also automatically configures the session and cache to the `array` driver while testing, meaning no session or cache data will be persisted while testing.

You are free to define other testing environment configuration values as necessary. The `testing` environment variables may be configured in your application's `phpunit.xml` file, but make sure to clear your configuration cache using the `config:clear` Artisan command before running your tests!

The `.env.testing` Environment File

In addition, you may create a `.env.testing` file in the root of your project. This file will be used instead of the `.env` file when running PHPUnit tests or executing Artisan commands with the `--env=testing` option.

Creating & Running Tests

To create a new test case, use the `make:test` Artisan command. By default, tests will be placed in the `tests/Feature` directory:

```
php artisan make:test UserTest
```

If you would like to create a test within the `tests/Unit` directory, you may use the `--unit` option when executing the `make:test` command:

```
php artisan make:test UserTest --unit
```

> TIP Test stubs may be customized using stub publishing

Once the test has been generated, you may define test methods as you normally would using PHPUnit. To run your tests, execute the `vendor/bin/phpunit` or `php artisan test` command from your terminal:

```
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_basic_test()
    {
        $this->assertTrue(true);
    }
}
```

NOTE If you define your own `setUp` / `tearDown` methods within a test class, be sure to call the respective `parent::setUp()` / `parent::tearDown()` methods on the parent class.

Artisan Test Runner

In addition to the `phpunit` command, you may use the `test` Artisan command to run your tests. The Artisan test runner provides verbose test reports in order to ease development and debugging:

```
php artisan test
```

Any arguments that can be passed to the `phpunit` command may also be passed to the Artisan `test` command:

```
php artisan test --testsuite=Feature --stop-on-failure
```

Testing

HTTP Tests

- Introduction
- Making Requests
  - Customizing Request Headers
  - Cookies
  - Session / Authentication
  - Debugging Responses
- Testing JSON APIs
- Testing File Uploads
- Testing Views
  - Rendering Blade & Components
- Available Assertions
  - Response Assertions
  - Authentication Assertions

Introduction

Laravel provides a very fluent API for making HTTP requests to your application and examining the responses. For example, take a look at the feature test defined below:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_a_basic_request()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

The `get` method makes a `GET` request into the application, while the `assertStatus` method asserts that the returned response should have the given HTTP status code. In addition to this simple assertion, Laravel also contains a variety of assertions for inspecting the response headers, content, JSON structure, and more.

Making Requests

To make a request to your application, you may invoke the `get`, `post`, `put`, `patch`, or `delete` methods within your test. These methods do not actually issue a "real" HTTP request to your application. Instead, the entire network request is simulated internally.

Instead of returning an `Illuminate\Http\Response` instance, test request methods return an instance of `Illuminate\Testing\TestResponse`, which provides a variety of helpful assertions that allow you to inspect your application's responses:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
```

```
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_a_basic_request()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

TIP For convenience, the CSRF middleware is automatically disabled when running tests.

Customizing Request Headers

You may use the `withHeaders` method to customize the request's headers before it is sent to the application. This method allows you to add any custom headers you would like to the request:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_interacting_with_headers()
    {
        $response = $this->withHeaders([
            'X-Header' => 'Value',
        ])->post('/user', ['name' => 'Sally']);

        $response->assertStatus(201);
    }
}
```

Cookies

You may use the `withCookie` or `withCookies` methods to set cookie values before making a request. The `withCookie` method accepts a cookie name and value as its two arguments, while the `withCookies` method accepts an array of name / value pairs:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_cookies()
    {
        $response = $this->withCookie('color', 'blue')->get('/');

        $response = $this->withCookies([
            'color' => 'blue',
            'name' => 'Taylor',
        ])->get('/');
    }
}
```

Session / Authentication

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

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_the_session()
    {
        $response = $this->withSession(['banned' => false])->get('/');
    }
}
```

Laravel's session is typically used to maintain state for the currently authenticated user. Therefore, the `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a model factory to generate and authenticate a user:

```
<?php

namespace Tests\Feature;

use App\Models\User;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_an_action_that_requires_authentication()
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
                         ->withSession(['banned' => false])
                         ->get('/');
    }
}
```

You may also specify which guard should be used to authenticate the given user by passing the guard name as the second argument to the `actingAs` method:

```
$this->actingAs($user, 'api')
```

Debugging Responses

After making a test request to your application, the `dump`, `dumpHeaders`, and `dumpSession` methods may be used to examine and debug the response contents:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_basic_test()
    {
        $response = $this->get('/');

        $response->dumpHeaders();

        $response->dumpSession();

        $response->dump();
```

```
    }
}
```

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `json`, `getJson`, `postJson`, `putJson`, `patchJson`, `deleteJson`, and `optionsJson` methods may be used to issue JSON requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/api/user` and assert that the expected JSON data was returned:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_making_an_api_request()
    {
        $response = $this->postJson('/api/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

In addition, JSON response data may be accessed as array variables on the response, making it convenient for you to inspect the individual values returned within a JSON response:

```
$this->assertTrue($response['created']);
```

> TIP The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

Asserting Exact JSON Matches

As previously mentioned, the `assertJson` method may be used to assert that a fragment of JSON exists within the JSON response. If you would like to verify that a given array **exactly matches** the JSON returned by your application, you should use the `assertExactJson` method:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

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

        $response
            ->assertStatus(201)
            ->assertExactJson([
                'created' => true,
            ]);
    }
```

```
}
```

Asserting On JSON Paths

If you would like to verify that the JSON response contains the given data at a specified path, you should use the `assertJsonPath` method:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

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

        $response
            ->assertStatus(201)
            ->assertJsonPath('team.owner.name', 'Darian');
    }
}
```

Testing File Uploads

The `Illuminate\Http\UploadedFile` class provides a `fake` method which may be used to generate dummy files or images for testing. This, combined with the `Storage` facade's `fake` method, greatly simplifies the testing of file uploads. For example, you may combine these two features to easily test an avatar upload form:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_avatars_can_be_uploaded()
    {
        Storage::fake('avatars');

        $file = UploadedFile::fake()->image('avatar.jpg');

        $response = $this->post('/avatar', [
            'avatar' => $file,
        ]);

        Storage::disk('avatars')->assertExists($file->hashName());
    }
}
```

If you would like to assert that a given file does not exist, you may use the `assertMissing` method provided by the `Storage` facade:

```
Storage::fake('avatars');

// ...

Storage::disk('avatars')->assertMissing('missing.jpg');
```

Fake File Customization

When creating files using the `fake` method provided by the `UploadedFile` class, you may specify the width, height, and size of the image (in kilobytes) in order to better test your application's validation rules:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

In addition to creating images, you may create files of any other type using the `create` method:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

If needed, you may pass a `$mimeType` argument to the method to explicitly define the MIME type that should be returned by the file:

```
UploadedFile::fake()->create(
    'document.pdf', $sizeInKilobytes, 'application/pdf'
);
```

Testing Views

Laravel also allows you to render a view without making a simulated HTTP request to the application. To accomplish this, you may call the `view` method within your test. The `view` method accepts the view name and an optional array of data. The method returns an instance of `Illuminate\Testing\TestView`, which offers several methods to conveniently make assertions about the view's contents:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_a_welcome_view_can_be_rendered()
    {
        $view = $this->view('welcome', ['name' => 'Taylor']);

        $view->assertSee('Taylor');
    }
}
```

The `TestView` class provides the following assertion methods: `assertSee`, `assertSeeInOrder`, `assertSeeText`, `assertSeeTextInOrder`, `assertDontSee`, and `assertDontSeeText`.

If needed, you may get the raw, rendered view contents by casting the `TestView` instance to a string:

```
$contents = (string) $this->view('welcome');
```

Sharing Errors

Some views may depend on errors shared in the global error bag provided by Laravel. To hydrate the error bag with error messages, you may use the `withViewErrors` method:

```
$view = $this->withViewErrors([
    'name' => ['Please provide a valid name.']
])->view('form');

$view->assertSee('Please provide a valid name.');
```

Rendering Blade & Components

If necessary, you may use the `blade` method to evaluate and render a raw Blade string. Like the `view` method, the `blade` method returns an instance of `Illuminate\Testing\TestView`:

```
$view = $this->blade(
    '<x-component :name="$name" />',
    ['name' => 'Taylor']
);

$view->assertSee('Taylor');
```

You may use the `component` method to evaluate and render a Blade component. Like the `view` method, the `component` method returns an instance of `Illuminate\Testing\TestView`:

```
$view = $this->component(Profile::class, ['name' => 'Taylor']);

$view->assertSee('Taylor');
```

Available Assertions

Response Assertions

Laravel's `Illuminate\Testing\TestResponse` class provides a variety of custom assertion methods that you may utilize when testing your application. These assertions may be accessed on the response that is returned by the `json`, `get`, `post`, `put`, and `delete` test methods:

assertCookie assertCookieExpired assertCookieNotExpired assertCookieMissing assertCreated assertDontSee assertDontSeeText assertExactJson assertForbidden assertHeader assertHeaderMissing assertJson assertJsonCount assertJsonFragment assertJsonMissing assertJsonMissingExact assertJsonMissingValidationErrors assertJsonPath assertJsonStructure assertJsonValidationErrors assertLocation assertNoContent assertNotFound assertOk assertPlainCookie assertRedirect assertSee assertSeeInOrder assertSeeText assertSeeTextInOrder assertSessionHas assertSessionHasInput assertSessionHasAll assertSessionHasErrors assertSessionHasErrorsIn assertSessionHasNoErrors assertSessionDoesntHaveErrors assertSessionMissing assertStatus assertSuccessful assertUnauthorized assertViewHas assertViewHasAll assertViewIs assertViewMissing

assertCookie

Assert that the response contains the given cookie:

```
$response->assertCookie($cookieName, $value = null);
```

assertCookieExpired

Assert that the response contains the given cookie and it is expired:

```
$response->assertCookieExpired($cookieName);
```

assertCookieNotExpired

Assert that the response contains the given cookie and it is not expired:

```
$response->assertCookieNotExpired($cookieName);
```

assertCookieMissing

Assert that the response does not contains the given cookie:

```
$response->assertCookieMissing($cookieName);
```

assertCreated

Assert that the response has a 201 HTTP status code:

```
$response->assertCreated();
```

assertDontSee

Assert that the given string is not contained within the response returned by the application. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertDontSee($value, $escaped = true);
```

assertDontSeeText

Assert that the given string is not contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. This method will pass the response content to the `strip_tags` PHP function before making the assertion:

```
$response->assertDontSeeText($value, $escaped = true);
```

assertExactJson

Assert that the response contains an exact match of the given JSON data:

```
$response->assertExactJson(array $data);
```

assertForbidden

Assert that the response has a forbidden (403) HTTP status code:

```
$response->assertForbidden();
```

assertHeader

Assert that the given header and value is present on the response:

```
$response->assertHeader($headerName, $value = null);
```

assertHeaderMissing

Assert that the given header is not present on the response:

```
$response->assertHeaderMissing($headerName);
```

assertJson

Assert that the response contains the given JSON data:

```
$response->assertJson(array $data, $strict = false);
```

The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

assertJsonCount

Assert that the response JSON has an array with the expected number of items at the given key:

```
$response->assertJsonCount($count, $key = null);
```

assertJsonFragment

Assert that the response contains the given JSON data anywhere in the response:

```
Route::get('/users', function () {
    return [
        'users' => [
            [
                'name' => 'Taylor Otwell',
            ],
        ],
    ];
});
```

```
$response->assertJsonFragment(['name' => 'Taylor Otwell']);
```

assertJsonMissing

Assert that the response does not contain the given JSON data:

```
$response->assertJsonMissing(array $data);
```

assertJsonMissingExact

Assert that the response does not contain the exact JSON data:

```
$response->assertJsonMissingExact(array $data);
```

assertJsonMissingValidationErrors

Assert that the response has no JSON validation errors for the given keys:

```
$response->assertJsonMissingValidationErrors($keys);
```

assertJsonPath

Assert that the response contains the given data at the specified path:

```
$response->assertJsonPath($path, array $data, $strict = true);
```

For example, if the JSON response returned by your application contains the following data:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the `name` property of the `user` object matches a given value like so:

```
$response->assertJsonPath('user.name', 'Steve Schoger');
```

assertJsonStructure

Assert that the response has a given JSON structure:

```
$response->assertJsonStructure(array $structure);
```

For example, if the JSON response returned by your application contains the following data:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the JSON structure matches your expectations like so:

```
$response->assertJsonStructure([
    'user' => [
        'name',
    ]
]);
```

assertJsonValidationErrors

Assert that the response has the given JSON validation errors for the given keys. This method should be used when asserting against responses where the validation errors are returned as a JSON structure instead of being flashed to the session:

```
$response->assertJsonValidationErrors(array $data);
```

assertLocation

Assert that the response has the given URI value in the `Location` header:

```
$response->assertLocation($uri);
```

assertNoContent

Assert that the response has the given HTTP status code and no content:

```
$response->assertNoContent($status = 204);
```

assertNotFound

Assert that the response has a not found (404) HTTP status code:

```
$response->assertNotFound();
```

assertOk

Assert that the response has a 200 HTTP status code:

```
$response->assertOk();
```

assertPlainCookie

Assert that the response contains the given unencrypted cookie:

```
$response->assertPlainCookie($cookieName, $value = null);
```

assertRedirect

Assert that the response is a redirect to the given URI:

```
$response->assertRedirect($uri);
```

assertSee

Assert that the given string is contained within the response. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertSee($value, $escaped = true);
```

assertSeeInOrder

Assert that the given strings are contained in order within the response. This assertion will automatically escape the given strings unless you pass a second argument of `false`:

```
$response->assertSeeInOrder(array $values, $escaped = true);
```

assertSeeText

Assert that the given string is contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. The response content will be passed to the `strip_tags` PHP function before the assertion is made:

```
$response->assertSeeText($value, $escaped = true);
```

assertSeeTextInOrder

Assert that the given strings are contained in order within the response text. This assertion will automatically escape the given strings unless you pass a second argument of `false`. The response content will be passed to the

`strip_tags` PHP function before the assertion is made:

```
$response->assertSeeTextInOrder(array $values, $escaped = true);
```

assertSessionHas

Assert that the session contains the given piece of data:

```
$response->assertSessionHas($key, $value = null);
```

assertSessionHasInput

Assert that the session has a given value in the flashed input array:

```
$response->assertSessionHasInput($key, $value = null);
```

assertSessionHasAll

Assert that the session contains a given array of key / value pairs:

```
$response->assertSessionHasAll(array $data);
```

For example, if your application's session contains `name` and `status` keys, you may assert that both exist and have the specified values like so:

```
$response->assertSessionHasAll([
    'name' => 'Taylor Otwell',
    'status' => 'active',
]);
```

assertSessionHasErrors

Assert that the session contains an error for the given `$keys`. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key). This method should be used when testing routes that flash validation errors to the session instead of returning them as a JSON structure:

```
$response->assertSessionHasErrors(
    array $keys, $format = null, $errorBag = 'default'
);
```

For example, to assert that the `name` and `email` field have validation error messages that were flashed to the session, you may invoke the `assertSessionHasErrors` method like so:

```
$response->assertSessionHasErrors(['name', 'email']);
```

Or, you may assert that a given field has a particular validation error message:

```
$response->assertSessionHasErrors([
    'name' => 'The given name was invalid.'
]);
```

assertSessionHasErrorsIn

Assert that the session contains an error for the given `$keys` within a specific error bag. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key), within the error bag:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

assertSessionHasNoErrors

Assert that the session has no validation errors:

```
$response->assertSessionHasNoErrors();
```

assertSessionDoesntHaveErrors

Assert that the session has no validation errors for the given keys:

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag = 'default');
```

assertSessionMissing

Assert that the session does not contain the given key:

```
$response->assertSessionMissing($key);
```

assertStatus

Assert that the response has a given HTTP status code:

```
$response->assertStatus($code);
```

assertSuccessful

Assert that the response has a successful (>= 200 and < 300) HTTP status code:

```
$response->assertSuccessful();
```

assertUnauthorized

Assert that the response has an unauthorized (401) HTTP status code:

```
$response->assertUnauthorized();
```

assertViewHas

Assert that the response view contains given a piece of data:

```
$response->assertViewHas($key, $value = null);
```

In addition, view data may be accessed as array variables on the response, allowing you to convenient inspect it:

```
$this->assertEquals('Taylor', $response['name']);
```

assertViewHasAll

Assert that the response view has a given list of data:

```
$response->assertViewHasAll(array $data);
```

This method may be used to assert that the view simply contains data matching the given keys:

```
$rseponse->assertViewHasAll([
    'name',
    'email',
]);
```

Or, you may assert that the view data is present and has specific values:

```
$rseponse->assertViewHasAll([
    'name' => 'Taylor Otwell',
    'email' => 'taylor@example.com,',
]);
```

assertViewIs

Assert that the given view was returned by the route:

```
$response->assertViewIs($value);
```

assertViewMissing

Assert that the given data key was not made available to the view returned in the application's response:

```
$response->assertViewMissing($key);
```

Authentication Assertions

Laravel also provides a variety of authentication related assertions that you may utilize within your application's feature tests. Note that these methods are invoked on the test class itself and not the `Illuminate\Testing\TestResponse` instance returned by methods such as `get` and `post`.

assertAuthenticated

Assert that a user is authenticated:

```
$this->assertAuthenticated($guard = null);
```

assertGuest

Assert that a user is not authenticated:

```
$this->assertGuest($guard = null);
```

assertAuthenticatedAs

Assert that a specific user is authenticated:

```
$this->assertAuthenticatedAs($user, $guard = null);
```

Testing

Console Tests

- Introduction
- Input / Output Expectations

Introduction

In addition to simplifying HTTP testing, Laravel provides a simple API for testing your application's custom console commands.

Input / Output Expectations

Laravel allows you to easily "mock" user input for your console commands using the `expectsQuestion` method. In addition, you may specify the exit code and text that you expect to be output by the console command using the `assertExitCode` and `expectsOutput` methods. For example, consider the following console command:

```
Artisan::command('question', function () {
    $name = $this->ask('What is your name?');

    $language = $this->choice('Which language do you prefer?', [
        'PHP',
        'Ruby',
        'Python',
    ]);

    $this->line('Your name is '.$name.' and you prefer '.$language.'.');
});
```

You may test this command with the following test which utilizes the `expectsQuestion`, `expectsOutput`, `doesntExpectOutput`, and `assertExitCode` methods:

```
/**
 * Test a console command.
 *
 * @return void
 */
public function test_console_command()
{
    $this->artisan('question')
         ->expectsQuestion('What is your name?', 'Taylor Otwell')
         ->expectsQuestion('Which language do you prefer?', 'PHP')
         ->expectsOutput('Your name is Taylor Otwell and you prefer PHP.')
         ->doesntExpectOutput('Your name is Taylor Otwell and you prefer Ruby.')
         ->assertExitCode(0);
}
```

Confirmation Expectations

When writing a command which expects a confirmation in the form of a "yes" or "no" answer, you may utilize the `expectsConfirmation` method:

```
$this->artisan('module:import')
    ->expectsConfirmation('Do you really wish to run this command?', 'no')
    ->assertExitCode(1);
```

Table Expectations

If your command displays a table of information using Artisan's `table` method, it can be cumbersome to write output expectations for the entire table. Instead, you may use the `expectsTable` method. This method accepts the table's headers as its first argument and the table's data as its second argument:

```
$this->artisan('users:all')
    ->expectsTable([
        'ID',
        'Email',
```

```
], [
    [1, 'taylor@example.com'],
    [2, 'abigail@example.com'],
]);
```

Testing

Laravel Dusk

- Introduction
- Installation
  - Managing ChromeDriver Installations
  - Using Other Browsers
- Getting Started
  - Generating Tests
  - Database Migrations
  - Running Tests
  - Environment Handling
- Browser Basics
  - Creating Browsers
  - Navigation
  - Resizing Browser Windows
  - Browser Macros
  - Authentication
  - Cookies
  - Executing JavaScript
  - Taking A Screenshot
  - Storing Console Output To Disk
  - Storing Page Source To Disk
- Interacting With Elements
  - Dusk Selectors
  - Text, Values, & Attributes
  - Interacting With Forms
  - Attaching Files
  - Pressing Buttons
  - Clicking Links
  - Using The Keyboard
  - Using The Mouse
  - JavaScript Dialogs
  - Scoping Selectors
  - Waiting For Elements
  - Scrolling An Element Into View
- Available Assertions
- Pages
  - Generating Pages
  - Configuring Pages
  - Navigating To Pages
  - Shorthand Selectors
  - Page Methods
- Components
  - Generating Components
  - Using Components
- Continuous Integration
  - Heroku CI
  - Travis CI
  - GitHub Actions

Introduction

Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your local computer. Instead, Dusk uses a standalone ChromeDriver installation. However, you are free to utilize any other Selenium compatible driver you wish.

Installation

To get started, you should add the `laravel/dusk` Composer dependency to your project:

```
composer require --dev laravel/dusk
```

> NOTE If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.

After installing the Dusk package, execute the `dusk:install` Artisan command. The `dusk:install` command will create a `tests/Browser` directory and an example Dusk test:

```
php artisan dusk:install
```

Next, set the `APP_URL` environment variable in your application's `.env` file. This value should match the URL you use to access your application in a browser.

> TIP If you are using Laravel Sail to manage your local development environment, please also consult the Sail documentation on configuring and running Dusk tests.

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is included with Laravel Dusk, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 86

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all

# Install the version of ChromeDriver that matches the detected version of Chrome / Chromium for your
OS...
php artisan dusk:chrome-driver --detect
```

> NOTE Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone ChromeDriver installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the ChromeDriver:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
/**
```

```
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Database Migrations

Most of the tests you write will interact with pages that retrieve data from your application's database; however, your Dusk tests should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which will not be applicable or available across HTTP requests. Instead, use the `DatabaseMigrations` trait, which re-migrates the database for each test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

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

> NOTE SQLite in-memory databases may not be used when executing Dusk tests. Since the browser executes within its own process, it will not be able to access the in-memory databases of other processes.

Running Tests

To run your browser tests, execute the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, such as allowing you to only run the tests for a given group:

```
php artisan dusk --group=foo
```

> TIP If you are using Laravel Sail to manage your local development environment, please consult the Sail documentation on configuring and running Dusk tests.

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular

system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class to reflect the correct port:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a `.env.dusk.local` file.

When running tests, Dusk will back-up your `.env` file and rename your Dusk environment to `.env`. Once the tests have completed, your `.env` file will be restored.

Browser Basics

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, you may call the `browse` method from within your Dusk test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function test_basic_example()
    {
        $user = User::factory()->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
```

```
            $browser->visit('/login')
                    ->type('email', $user->email)
                    ->type('password', 'password')
                    ->press('Login')
                    ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a closure. A browser instance will automatically be passed to this closure by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, simply add more browser arguments to the signature of the closure given to the `browse` method:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home')
          ->waitForText('Message');

    $second->loginAs(User::find(2))
           ->visit('/home')
           ->waitForText('Message')
           ->type('message', 'Hey Taylor')
           ->press('Send');

    $first->waitForText('Hey Taylor')
          ->assertSee('Jeffrey Way');
});
```

Navigation

The `visit` method may be used to navigate to a given URI within your application:

```
$browser->visit('/login');
```

You may use the `visitRoute` method to navigate to a named route:

```
$browser->visitRoute('login');
```

You may navigate "back" and "forward" using the `back` and `forward` methods:

```
$browser->back();

$browser->forward();
```

You may use the `refresh` method to refresh the page:

```
$browser->refresh();
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

The `fitContent` method will resize the browser window to match the size of its content:

```
$browser->fitContent();
```

When a test fails, Dusk will automatically resize the browser to fit the content prior to taking a screenshot. You

may disable this feature by calling the `disableFitOnFailure` method within your test:

```
$browser->disableFitOnFailure();
```

You may use the `move` method to move the browser window to a different position on your screen:

```
$browser->move($x = 100, $y = 100);
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     *
     * @return void
     */
    public function boot()
    {
        Browser::macro('scrollToElement', function ($element = null) {
            $this->script("$('html, body').animate({ scrollTop: $('$element').offset().top }, 0);");

            return $this;
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Browser` instance:

```
$this->browse(function ($browser) use ($user) {
    $browser->visit('/pay')
            ->scrollToElement('#credit-card-details')
            ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with your application's login screen during every test. The `loginAs` method accepts a primary key associated with your authenticatable model or an authenticatable model instance:

```
use App\Models\User;

$this->browse(function ($browser) {
    $browser->loginAs(User::find(1))
          ->visit('/home');
});
```

NOTE After using the `loginAs` method, the user session will be maintained for all tests within the file.

Cookies

You may use the `cookie` method to get or set an encrypted cookie's value. By default, all of the cookies created by Laravel are encrypted:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

You may use the `plainCookie` method to get or set an unencrypted cookie's value:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

You may use the `deleteCookie` method to delete the given cookie:

```
$browser->deleteCookie('name');
```

Executing JavaScript

You may use the `script` method to execute arbitrary JavaScript statements within the browser:

```
$output = $browser->script('document.documentElement.scrollTop = 0');

$output = $browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);
```

Taking A Screenshot

You may use the `screenshot` method to take a screenshot and store it with the given filename. All screenshots will be stored within the `tests/Browser/screenshots` directory:

```
$browser->screenshot('filename');
```

Storing Console Output To Disk

You may use the `storeConsoleLog` method to write the current browser's console output to disk with the given filename. Console output will be stored within the `tests/Browser/console` directory:

```
$browser->storeConsoleLog('filename');
```

Storing Page Source To Disk

You may use the `storeSource` method to write the current page's source to disk with the given filename. The page source will be stored within the `tests/Browser/source` directory:

```
$browser->storeSource('filename');
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, when interacting with a Dusk browser, prefix the selector with `@` to manipulate the attached element within your test:

```
// HTML...

<button dusk="login-button">Login</button>

// Test...
```

```
$browser->click('@login-button');
```

Text, Values, & Attributes

Retrieving & Setting Values

Dusk provides several methods for interacting with the current display text, value, and attributes of elements on the page. For example, to get the "value" of an element that matches a given CSS or Dusk selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

You may use the `inputValue` method to get the "value" of an input element that has a given field name:

```
$value = $browser->inputValue('field');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve the value of an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Interacting With Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an `input` or `textarea` field with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
        ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

You can instruct Dusk to type slowly using the `typeSlowly` method. By default, Dusk will pause for 100 milliseconds between key presses. To customize the amount of time between key presses, you may pass the appropriate number of milliseconds as the third argument to the method:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');

$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

You may use the `appendSlowly` method to append text slowly:

```
$browser->type('tags', 'foo')
        ->appendSlowly('tags', ', bar, baz');
```

Dropdowns

To select a value available on a `select` element, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second argument:

```
$browser->select('size');
```

Checkboxes

To "check" a checkbox input, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');
```

The `uncheck` method may be used to "uncheck" a checkbox input:

```
$browser->uncheck('terms');
```

Radio Buttons

To "select" a `radio` input option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `radio` input with matching `name` and `value` attributes:

```
$browser->radio('size', 'large');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `file` input with a matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```

> NOTE The attach function requires the `Zip` PHP extension to be installed and enabled on your server.

Pressing Buttons

The `press` method may be used to click a button element on the page. The first argument given to the `press` method may be either the display text of the button or a CSS / Dusk selector:

```
$browser->press('Login');
```

When submitting forms, many application's disable the form's submission button after it are pressed and then re-enable the button when the form submission's HTTP request is complete. To press a button and wait for the button to be re-enabled, you may use the `pressAndWaitFor` method:

```
// Press the button and wait a maximum of 5 seconds for it to be enabled...
$browser->pressAndWaitFor('Save');

// Press the button and wait a maximum of 1 second for it to be enabled...
$browser->pressAndWaitFor('Save', 1);
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

You may use the `seeLink` method to determine if a link with the given display text is visible on the page:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```

> NOTE These methods interact with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Using The Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may instruct Dusk to hold modifier keys while entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `swift` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Another valuable use case for the `keys` method is sending a "keyboard shortcut" combination to the primary CSS selector for your application:

```
$browser->keys('.app', ['{command}', 'j']);
```

> TIP All modifier keys such as `{command}` are wrapped in `{}` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be found on GitHub.

Using The Mouse

Clicking On Elements

The `click` method may be used to click on an element matching the given CSS or Dusk selector:

```
$browser->click('.selector');
```

The `clickAtXPath` method may be used to click on an element matching the given XPath expression:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

The `clickAtPoint` method may be used to click on the topmost element at a given pair of coordinates relative to the viewable area of the browser:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

The `doubleClick` method may be used to simulate the double click of a mouse:

```
$browser->doubleClick();
```

The `rightClick` method may be used to simulate the right click of a mouse:

```
$browser->rightClick();
```

```
$browser->rightClick('.selector');
```

The `clickAndHold` method may be used to simulate a mouse button being clicked and held down. A subsequent call to the `releaseMouse` method will undo this behavior and release the mouse button:

```
$browser->clickAndHold()
        ->pause(1000)
        ->releaseMouse();
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given CSS or Dusk selector:

```
$browser->mouseover('.selector');
```

Drag & Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', $pixels = 10);
$browser->dragRight('.selector', $pixels = 10);
$browser->dragUp('.selector', $pixels = 10);
$browser->dragDown('.selector', $pixels = 10);
```

Finally, you may drag an element by a given offset:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs. For example, you may use the `waitForDialog` method to wait for a JavaScript dialog to appear. This method accepts an optional argument indicating how many seconds to wait for the dialog to appear:

```
$browser->waitForDialog($seconds = null);
```

The `assertDialogOpened` method may be used to assert that a dialog has been displayed and contains the given message:

```
$browser->assertDialogOpened('Dialog message');
```

If the JavaScript dialog contains a prompt, you may use the `typeInDialog` method to type a value into the prompt:

```
$browser->typeInDialog('Hello World');
```

To close an open JavaScript dialog by clicking the "OK" button, you may invoke the `acceptDialog` method:

```
$browser->acceptDialog();
```

To close an open JavaScript dialog by clicking the "Cancel" button, you may invoke the `dismissDialog` method:

```
$browser->dismissDialog();
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to accomplish this. All operations performed within the closure given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

You may occasionally need to execute assertions outside of the current scope. You may use the `elsewhere` method to accomplish this:

```
 $browser->with('.table', function ($table) {
    // Current scope is `body .table`...

    $browser->elsewhere('.page-title', function ($title) {
```

```
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });
});
```

Waiting For Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to be visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you just need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

Waiting For Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS or Dusk selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

You may also wait until the element matching the given selector contains the given text:

```
// Wait a maximum of five seconds for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World');

// Wait a maximum of one second for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

You may also wait until the element matching the given selector is missing from the page:

```
// Wait a maximum of five seconds until the selector is missing...
$browser->waitUntilMissing('.selector');

// Wait a maximum of one second until the selector is missing...
$browser->waitUntilMissing('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for an element to appear that matches a given selector and then interact with the element. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used to accomplish this. All element operations performed within the given closure will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
          ->press('OK');
});
```

Waiting For Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
```

```
$browser->waitForText('Hello World', 1);
```

You may use the `waitUntilMissingText` method to wait until the displayed text has been removed from the page:

```
// Wait a maximum of five seconds for the text to be removed...
$browser->waitUntilMissingText('Hello World');

// Wait a maximum of one second for the text to be removed...
$browser->waitUntilMissingText('Hello World', 1);
```

Waiting For Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting On The Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to make assertions after a page has been reloaded, use the `waitForReload` method:

```
$browser->click('.some-action')
        ->waitForReload()
        ->assertSee('something');
```

Waiting On JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitUntil` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

Waiting On Vue Expressions

The `waitUntilVue` and `waitUntilVueIsNot` methods may be used to wait until a Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitUntilVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitUntilVueIsNot('user.name', null, '@user');
```

Waiting With A Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given closure to return `true`. The `waitUsing` method accepts the maximum number of

seconds to wait, the interval at which the closure should be evaluated, the closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Scrolling An Element Into View

Sometimes you may not be able to click on an element because it is outside of the viewable area of the browser. The `scrollIntoView` method will scroll the browser window until the element at the given selector is within the view:

```
$browser->scrollIntoView('.selector')
        ->click('.selector');
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

assertTitle assertTitleContains assertUrlIs assertSchemeIs assertSchemeIsNot assertHostIs assertHostIsNot assertPortIs assertPortIsNot assertPathBeginsWith assertPathIs assertPathIsNot assertRouteIs assertQueryStringHas assertQueryStringMissing assertFragmentIs assertFragmentBeginsWith assertFragmentIsNot assertHasCookie assertHasPlainCookie assertCookieMissing assertPlainCookieMissing assertCookieValue assertPlainCookieValue assertSee assertDontSee assertSeeIn assertDontSeeIn assertSeeAnythingIn assertSeeNothingIn assertScript assertSourceHas assertSourceMissing assertSeeLink assertDontSeeLink assertInputValue assertInputValueIsNot assertChecked assertNotChecked assertRadioSelected assertRadioNotSelected assertSelected assertNotSelected assertSelectHasOptions assertSelectMissingOptions assertSelectHasOption assertSelectMissingOption assertValue assertAttribute assertAriaAttribute assertDataAttribute assertVisible assertPresent assertMissing assertDialogOpened assertEnabled assertDisabled assertButtonEnabled assertButtonDisabled assertFocused assertNotFocused assertAuthenticated assertGuest assertAuthenticatedAs assertVue assertVueIsNot assertVueContains assertVueDoesNotContain

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith('/home');
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the URL's current hash fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the URL's current hash fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the URL's current hash fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given encrypted cookie is present:

```
$browser->assertHasCookie($name);
```

assertHasPlainCookie

Assert that the given unencrypted cookie is present:

```
$browser->assertHasPlainCookie($name);
```

assertCookieMissing

Assert that the given encrypted cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertPlainCookieMissing

Assert that the given unencrypted cookie is not present:

```
$browser->assertPlainCookieMissing($name);
```

assertCookieValue

Assert that an encrypted cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSeeAnythingIn

Assert that any text is present within the selector:

```
$browser->assertSeeAnythingIn($selector);
```

assertSeeNothingIn

Assert that no text is present within the selector:

```
$browser->assertSeeNothingIn($selector);
```

assertScript

Assert that the given JavaScript expression evaluates to the given value:

```
$browser->assertScript('window.isLoaded')
        ->assertScript('document.readyState', 'complete');
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValueIsNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertSelectMissingOption

Assert that the given value is not available to be selected:

```
$browser->assertSelectMissingOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertAttribute

Assert that the element matching the given selector has the given value in the provided attribute:

```
$browser->assertAttribute($selector, $attribute, $value);
```

assertAriaAttribute

Assert that the element matching the given selector has the given value in the provided aria attribute:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

For example, given the markup `<button aria-label="Add"></button>`, you may assert against the `aria-label` attribute like so:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

assertDataAttribute

Assert that the element matching the given selector has the given value in the provided data attribute:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

For example, given the markup `<tr id="row-1" data-content="attendees"></tr>`, you may assert against the `data-label` attribute like so:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present:

```
$browser->assertPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertAuthenticated

Assert that the user is authenticated:

```
$browser->assertAuthenticated();
```

assertGuest

Assert that the user is not authenticated:

```
$browser->assertGuest();
```

assertAuthenticatedAs

Assert that the user is authenticated as the given user:

```
$browser->assertAuthenticatedAs($user);
```

assertVue

Dusk even allows you to make assertions on the state of Vue component data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/**
 * A basic Vue test example.
 *
 * @return void
 */
public function testVue()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
                ->assertVue('user.name', 'Taylor', '@profile-component');
    });
}
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

assertVueDoesNotContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Dusk Pages allow you to define expressive actions that may then be performed on a given page via a single method. Pages also allow you to define short-cuts to common selectors for your application or for a single page.

Generating Pages

To generate a page object, execute the `dusk:page` Artisan command. All page objects will be placed in your application's `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be discussed in more detail below.

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**
 * Get the URL for the page.
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. It is not actually necessary to place anything within this method; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

Navigating To Pages

Once a page has been defined, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
        ->clickLink('Create Playlist')
        ->on(new CreatePlaylist)
        ->assertSee('@create');
```

Shorthand Selectors

The `elements` method within page classes allow you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
```

```
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Once the shortcut has been defined, you may use the shorthand selector anywhere you would typically use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  string  $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
                ->check('share')
                ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed as the first argument to custom page methods:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, execute the `dusk:component` Artisan command. New components are placed in the `tests/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  int  $year
     * @param  int  $month
     * @param  int  $day
     * @return void
     */
    public function selectDate(Browser $browser, $year, $month, $day)
    {
        $browser->click('@date-field')
                ->within('@year-list', function ($browser) use ($year) {
                    $browser->click($year);
                })
```

```
                ->within('@month-list', function ($browser) use ($month) {
                    $browser->click($month);
                })
                ->within('@day-list', function ($browser) use ($day) {
                    $browser->click($day);
                });
    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```
<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->within(new DatePicker, function ($browser) {
                        $browser->selectDate(2019, 1, 30);
                    })
                    ->assertSee('January');
        });
    }
}
```

Continuous Integration

> NOTE Most Dusk continuous integration configurations expect your Laravel application to be served using the built-in PHP development server on port 8000. Therefore, before continuing, you should ensure that your continous integration environment has an `APP_URL` environment variable value of `http://127.0.0.1:8000`.

Heroku CI

To run Dusk tests on Heroku CI, add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &' &&
nohup bash -c 'php artisan serve > /dev/null 2>&1 &' && php artisan dusk"
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on Travis CI, use the following `.travis.yml` configuration. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost &
  - php artisan serve &

script:
  - php artisan dusk
```

GitHub Actions

If you are using Github Actions to run your Dusk tests, you may use the following configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
name: CI
on: [push]
jobs:

  dusk-php:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: |
          sudo systemctl start mysql
          mysql --user="root" --password="root" -e "CREATE DATABASE 'my-database' character set UTF8mb4
collate utf8mb4_bin;"
      - name: Install Composer Dependencies
        run: composer install --no-progress --no-suggest --prefer-dist --optimize-autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver `/opt/google/chrome/chrome --version | cut -d " " -f3 | cut
-d "." -f1`
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux &
      - name: Run Laravel Server
        run: php artisan serve &
      - name: Run Dusk Tests
        env:
          APP_URL: "http://127.0.0.1:8000"
        run: php artisan dusk
      - name: Upload Screenshots
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: screenshots
          path: tests/Browser/screenshots
      - name: Upload Console Logs
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: console
          path: tests/Browser/console
```

Testing

Database Testing

- Introduction
  - Resetting The Database After Each Test
- Defining Model Factories
  - Concept Overview
  - Generating Factories
  - Factory States
  - Factory Callbacks
- Creating Models Using Factories
  - Instantiating Models
  - Persisting Models
  - Sequences
- Factory Relationships
  - Has Many Relationships
  - Belongs To Relationships
  - Many To Many Relationships
  - Polymorphic Relationships
  - Defining Relationships Within Factories
- Running Seeders
- Available Assertions

Introduction

Laravel provides a variety of helpful tools and assertions to make it easier to test your database driven applications. In addition, Laravel model factories and seeders make it painless to create test database records using your application's Eloquent models and relationships. We'll discuss all of these powerful features in the following documentation.

Resetting The Database After Each Test

Before proceeding much further, let's discuss how to reset your database after each of your tests so that data from a previous test does not interfere with subsequent tests. Laravel's included `Illuminate\Foundation\Testing\RefreshDatabase` trait will take care of this for you. Simply use the trait on your test class:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_basic_example()
    {
        $response = $this->get('/');

        // ...
    }
}
```

Defining Model Factories

Concept Overview

First, let's talk about Eloquent 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 see an example of how to write a factory, take a look at the `database/factories/UserFactory.php` file in your application. This factory is included with all new Laravel applications and contains the following factory definition:

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = User::class;

    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'name' => $this->faker->name,
            'email' => $this->faker->unique()->safeEmail,
            'email_verified_at' => now(),
            'password' => '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }
}
```

As you can see, in their most basic form, factories are classes that extend Laravel's base factory class and define a `model` property and `definition` method. The `definition` method returns the default set of attribute values that should be applied when creating a model using the factory.

Via the `faker` property, factories have access to the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

> TIP You can set your application's Faker locale by adding a `faker_locale` option to your `config/app.php` configuration file.

Generating Factories

To create a factory, execute the `make:factory` Artisan command:

```
php artisan make:factory PostFactory
```

The new factory class will be placed in your `database/factories` directory.

The `--model` option may be used to indicate the name of the model created by the factory. This option will pre-fill the generated factory file with the given model:

```
php artisan make:factory PostFactory --model=Post
```

Factory States

State manipulation methods allow you to define discrete modifications that can be applied to your model

factories in any combination. For example, your `Database\Factories\UserFactory` factory might contain a `suspended` state method that modifies one of its default attribute values.

State transformation methods typically call the `state` method provided by Laravel's base factory class. The `state` method accepts a closure which will receive the array of raw attributes defined for the factory and should return an array of attributes to modify:

```
/**
 * Indicate that the user is suspended.
 *
 * @return \Illuminate\Database\Eloquent\Factories\Factory
 */
public function suspended()
{
    return $this->state(function (array $attributes) {
        return [
            'account_status' => 'suspended',
        ];
    });
}
```

Factory Callbacks

Factory callbacks are registered using the `afterMaking` and `afterCreating` methods and allow you to perform additional tasks after making or creating a model. You should register these callbacks by defining a `configure` method on your factory class. This method will be automatically called by Laravel when the factory is instantiated:

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = User::class;

    /**
     * Configure the model factory.
     *
     * @return $this
     */
    public function configure()
    {
        return $this->afterMaking(function (User $user) {
            //
        })->afterCreating(function (User $user) {
            //
        });
    }

    // ...
}
```

Creating Models Using Factories

Instantiating Models

Once you have defined your factories, you may use the static `factory` method provided to your models by the `Illuminate\Database\Eloquent\Factories\HasFactory` trait in order to instantiate a factory instance for that model. Let's take a look at a few examples of creating models. First, we'll use the `make` method to create models without persisting them to the database:

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

```
public function test_models_can_be_instantiated()
{
    $user = User::factory()->make();

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

You may create a collection of many models using the `count` method:

```
$users = User::factory()->count(3)->make();
```

Applying States

You may also apply any of your states to the models. If you would like to apply multiple state transformations to the models, you may simply call the state transformation methods directly:

```
$users = User::factory()->count(5)->suspended()->make();
```

Overriding Attributes

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

```
$user = User::factory()->make([
    'name' => 'Abigail Otwell',
]);
```

Alternatively, the `state` method may be called directly on the factory instance to perform an inline state transformation:

```
$user = User::factory()->state([
    'name' => 'Abigail Otwell',
])->make();
```

TIP Mass assignment protection is automatically disabled when creating models using factories.

Connecting Factories & Models

The `HasFactory` trait's `factory` method will use conventions to determine the proper factory for the model. Specifically, the method will look for a factory in the `Database\Factories` namespace that has a class name matching the model name and is suffixed with `Factory`. If these conventions do not apply to your particular application or factory, you may overwrite the `newFactory` method on your model to return an instance of the model's corresponding factory directly:

```
use Database\Factories\Administration\FlightFactory;

/**
 * Create a new factory instance for the model.
 *
 * @return \Illuminate\Database\Eloquent\Factories\Factory
 */
protected static function newFactory()
{
    return FlightFactory::new();
}
```

Persisting Models

The `create` method instantiates model instances and persists them to the database using Eloquent's `save` method:

```
use App\Models\User;

public function test_models_can_be_persisted()
{
    // Create a single App\Models\User instance...
    $user = User::factory()->create();

    // Create three App\Models\User instances...
```

```
    $users = User::factory()->count(3)->create();

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

You may override the factory's default model attributes by passing an array of attributes to the `create` method:

```
$user = User::factory()->create([
    'name' => 'Abigail',
]);
```

Sequences

Sometimes you may wish to alternate the value of a given model attribute for each created model. You may accomplish this by defining a state transformation as a sequence. For example, you may wish to alternate the value of an `admin` column between `Y` and `N` for each created user:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
                ->count(10)
                ->state(new Sequence(
                    ['admin' => 'Y'],
                    ['admin' => 'N'],
                ))
                ->create();
```

In this example, five users will be created with an `admin` value of `Y` and five users will be created with an `admin` value of `N`.

Factory Relationships

Has Many Relationships

Next, let's explore building Eloquent model relationships using Laravel's fluent factory methods. First, let's assume our application has an `App\Models\User` model and an `App\Models\Post` model. Also, let's assume that the `User` model defines a `hasMany` relationship with `Post`. We can create a user that has three posts using the `has` method provided by the Laravel's factories. The `has` method accepts a factory instance:

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

$user = User::factory()
            ->has(Post::factory()->count(3))
            ->create();
```

By convention, when passing a `Post` model to the `has` method, Laravel will assume that the `User` model must have a `posts` method that defines the relationship. If necessary, you may explicitly specify the name of the relationship that you would like to manipulate:

```
$user = User::factory()
            ->has(Post::factory()->count(3), 'posts')
            ->create();
```

Of course, you may perform state manipulations on the related models. In addition, you may pass a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
            ->has(
                Post::factory()
                        ->count(3)
                        ->state(function (array $attributes, User $user) {
                            return ['user_type' => $user->type];
                        })
            )
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to build relationships. For example, the following example will use convention to determine that the related models should be created via a `posts` relationship method on the `User` model:

```
$user = User::factory()
            ->hasPosts(3)
            ->create();
```

When using magic methods to create factory relationships, you may pass an array of attributes to override on the related models:

```
$user = User::factory()
            ->hasPosts(3, [
                'published' => false,
            ])
            ->create();
```

You may provide a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
            ->hasPosts(3, function (array $attributes, User $user) {
                return ['user_type' => $user->type];
            })
            ->create();
```

Belongs To Relationships

Now that we have explored how to build "has many" relationships using factories, let's explore the inverse of the relationship. The `for` method may be used to define the parent model that factory created models belong to. For example, we can create three `App\Models\Post` model instances that belong to a single user:

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

$posts = Post::factory()
            ->count(3)
            ->for(User::factory()->state([
                'name' => 'Jessica Archer',
            ]))
            ->create();
```

If you already have a parent model instance that should be associated with the models you are creating, you may pass the model instance to the `for` method:

```
$user = User::factory()->create();

$posts = Post::factory()
            ->count(3)
            ->for($user)
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define "belongs to" relationships. For example, the following example will use convention to determine that the three posts should belong to the `user` relationship on the `Post` model:

```
$posts = Post::factory()
            ->count(3)
            ->forUser([
                'name' => 'Jessica Archer',
            ])
            ->create();
```

Many To Many Relationships

Like has many relationships, "many to many" relationships may be created using the `has` method:

```
use App\Models\Role;
use App\Models\User;
```

```
$user = User::factory()
            ->has(Role::factory()->count(3))
            ->create();
```

Pivot Table Attributes

If you need to define attributes that should be set on the pivot / intermediate table linking the models, you may use the `hasAttached` method. This method accepts an array of pivot table attribute names and values as its second argument:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
            ->hasAttached(
                Role::factory()->count(3),
                ['active' => true]
            )
            ->create();
```

You may provide a closure based state transformation if your state change requires access to the related model:

```
$user = User::factory()
            ->hasAttached(
                Role::factory()
                    ->count(3)
                    ->state(function (array $attributes, User $user) {
                        return ['name' => $user->name.' Role'];
                    }),
                ['active' => true]
            )
            ->create();
```

If you already have model instances that you would like to attached to the models you are creating, you may pass the model instances to the `hasAttached` method. In this example, the same three roles will be attached to all three users:

```
$roles = Role::factory()->count(3)->create();

$user = User::factory()
            ->count(3)
            ->hasAttached($roles, ['active' => true])
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define many to many relationships. For example, the following example will use convention to determine that the related models should be created via a `roles` relationship method on the `User` model:

```
$user = User::factory()
            ->hasRoles(1, [
                'name' => 'Editor'
            ])
            ->create();
```

Polymorphic Relationships

Polymorphic relationships may also be created using factories. Polymorphic "morph many" relationships are created in the same way as typical "has many" relationships. For example, if a `App\Models\Post` model has a `morphMany` relationship with a `App\Models\Comment` model:

```
use App\Models\Post;

$post = Post::factory()->hasComments(3)->create();
```

Morph To Relationships

Magic methods may not be used to create `morphTo` relationships. Instead, the `for` method must be used directly and the name of the relationship must be explicitly provided. For example, imagine that the `Comment` model has a `commentable` method that defines a `morphTo` relationship. In this situation, we may create three comments that belong to a single post by using the `for` method directly:

```
$comments = Comment::factory()->count(3)->for(
    Post::factory(), 'commentable'
)->create();
```

Polymorphic Many To Many Relationships

Polymorphic "many to many" (`morphToMany` / `morphedByMany`) relationships may be created just like non-polymorphic "many to many" relationships:

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
            ->hasAttached(
                Tag::factory()->count(3),
                ['public' => true]
            )
            ->create();
```

Of course, the magic `has` method may also be used to create polymorphic "many to many" relationships:

```
$videos = Video::factory()
            ->hasTags(3, ['public' => true])
            ->create();
```

Defining Relationships Within Factories

To define a relationship within your model factory, you will typically assign a new factory instance to the foreign key of the relationship. This is normally done for the "inverse" relationships such as `belongsTo` and `morphTo` relationships. For example, if you would like to create a new user when creating a post, you may do the following:

```
use App\Models\User;

/**
 * Define the model's default state.
 *
 * @return array
 */
public function definition()
{
    return [
        'user_id' => User::factory(),
        'title' => $this->faker->title,
        'content' => $this->faker->paragraph,
    ];
}
```

If the relationship's columns depend on the factory that defines it you may assign a closure to an attribute. The closure will receive the factory's evaluated attribute array:

```
/**
 * Define the model's default state.
 *
 * @return array
 */
public function definition()
{
    return [
        'user_id' => User::factory(),
        'user_type' => function (array $attributes) {
            return User::find($attributes['user_id'])->type;
        },
        'title' => $this->faker->title,
        'content' => $this->faker->paragraph,
    ];
}
```

Running Seeders

If you would like to use database seeders to populate your database during a feature test, you may invoke the `seed` method. By default, the `seed` method will execute the `DatabaseSeeder`, which should execute all of your other seeders. Alternatively, you pass a specific seeder class name to the `seed` method:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use OrderStatusSeeder;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Test creating a new order.
     *
     * @return void
     */
    public function test_orders_can_be_created()
    {
        // Run the DatabaseSeeder...
        $this->seed();

        // Run a specific seeder...
        $this->seed(OrderStatusSeeder::class);

        // ...
    }
}
```

Alternatively, you may instruct the `RefreshDatabase` trait to automatically seed the database before each test. You may accomplish this by defining a `$seed` property on your test class:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Indicates whether the database should be seeded before each test.
     *
     * @var bool
     */
    protected $seed = true;

    // ...
}
```

Available Assertions

Laravel provides several database assertions for your PHPUnit feature tests. We'll discuss each of these assertions below.

assertDatabaseCount

Assert that a table in the database contains the given number of records:

```
$this->assertDatabaseCount('users', 5);
```

assertDatabaseHas

Assert that a table in the database contains records matching the given key / value query constraints:

```
$this->assertDatabaseHas('users', [
    'email' => 'sally@example.com',
]);
```

assertDatabaseMissing

Assert that a table in the database does not contain records matching the given key / value query constraints:

```
$this->assertDatabaseMissing('users', [
    'email' => 'sally@example.com',
]);
```

assertDeleted

The `assertDeleted` asserts that a given Eloquent model has been deleted from the database:

```
use App\Models\User;

$user = User::find(1);

$user->delete();

$this->assertDeleted($user);
```

The `assertSoftDeleted` method may be used to assert a given Eloquent model has been "soft deleted":

```
$this->assertSoftDeleted($user);
```

Testing

Mocking

- Introduction
- Mocking Objects
- Mocking Facades
  - Facade Spies
- Bus Fake
  - Job Chains
  - Job Batches
- Event Fake
  - Scoped Event Fakes
- HTTP Fake
- Mail Fake
- Notification Fake
- Queue Fake
  - Job Chains
- Storage Fake
- Interacting With Time

Introduction

When testing Laravel applications, you may wish to "mock" certain aspects of your application so they are not actually executed during a given test. For example, when testing a controller that dispatches an event, you may wish to mock the event listeners so they are not actually executed during the test. This allows you to only test the controller's HTTP response without worrying about the execution of the event listeners since the event listeners can be tested in their own test case.

Laravel provides helpful methods for mocking events, jobs, and other facades out of the box. These helpers primarily provide a convenience layer over Mockery so you do not have to manually make complicated Mockery method calls.

Mocking Objects

When mocking an object that is going to be injected into your application via Laravel's service container, you will need to bind your mocked instance into the container as an `instance` binding. This will instruct the container to use your mocked instance of the object instead of constructing the object itself:

```
use App\Service;
use Mockery;
use Mockery\MockInterface;

public function test_something_can_be_mocked()
{
    $this->instance(
        Service::class,
        Mockery::mock(Service::class, function (MockInterface $mock) {
            $mock->shouldReceive('process')->once();
        })
    );
}
```

In order to make this more convenient, you may use the `mock` method that is provided by Laravel's base test case class. For example, the following example is equivalent to the example above:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->mock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

You may use the `partialMock` method when you only need to mock a few methods of an object. The methods

that are not mocked will be executed normally when called:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->partialMock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

Similarly, if you want to spy on an object, Laravel's base test case class offers a `spy` method as a convenient wrapper around the `Mockery::spy` method. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

```
use App\Service;

$spy = $this->spy(Service::class);

// ...

$spy->shouldHaveReceived('process')
```

Mocking Facades

Unlike traditional static method calls, facades (including real-time facades) may be mocked. This provides a great advantage over traditional static methods and grants you the same testability that you would have if you were using traditional dependency injection. When testing, you may often want to mock a call to a Laravel facade that occurs in one of your controllers. For example, consider the following controller action:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

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

        //
    }
}
```

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

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

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

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

```
        // ...
    }
}
```

NOTE You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP testing methods such as `get` and `post` when running your test. Likewise, instead of mocking the `Config` facade, call the `Config::set` method in your tests.

Facade Spies

If you would like to spy on a facade, you may call the `spy` method on the corresponding facade. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

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

public function test_values_are_be_stored_in_cache()
{
    Cache::spy();

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

    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->once()->with('name', 'Taylor', 10);
}
```

Bus Fake

When testing code that dispatches jobs, you typically want to assert that a given job was dispatched but not actually queue or execute the job. This is because the job's execution can normally be tested in a separate test class.

You may use the `Bus` facade's `fake` method to prevent jobs from being dispatched to the queue. Then, after executing the code under test, you may inspect which jobs the application attempted to dispatch using the `assertDispatched` and `assertNotDispatched` methods:

```
<?php

namespace Tests\Feature;

use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Bus;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Bus::fake();

        // Perform order shipping...

        // Assert that a job was dispatched...
        Bus::assertDispatched(ShipOrder::class);

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}
```

You may pass a closure to the `assertDispatched` or `assertNotDispatched` methods in order to assert that a job was dispatched that passes a given "truth test". If at least one job was dispatched that passes the given truth test then the assertion will be successful. For example, you may wish to assert that a job was dispatched for a specific order:

```
Bus::assertDispatched(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

Job Chains

The `Bus` facade's `assertChained` method may be used to assert that a chain of jobs was dispatched. The `assertChained` method accepts an array of chained jobs as its first argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;

Bus::assertChained([
    ShipOrder::class,
    RecordShipment::class,
    UpdateInventory::class
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Bus::assertChained([
    new ShipOrder,
    new RecordShipment,
    new UpdateInventory,
]);
```

Job Batches

The `Bus` facade's `assertBatched` method may be used to assert that a batch of jobs was dispatched. The closure given to the `assertBatched` method receives an instance of `Illuminate\Bus\PendingBatch`, which may be used to inspect the jobs within the batch:

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
           $batch->jobs->count() === 10;
});
```

Event Fake

When testing code that dispatches events, you may wish to instruct Laravel to not actually execute the event's listeners. Using the `Event` facade's `fake` method, you may prevent listeners from executing, execute the code under test, and then assert which events were dispatched by your application using the `assertDispatched` and `assertNotDispatched` methods:

```
<?php

namespace Tests\Feature;

use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function test_orders_can_be_shipped()
    {
        Event::fake();

        // Perform order shipping...

        // Assert that an event was dispatched...
```

```
        Event::assertDispatched(OrderShipped::class);

        // Assert an event was dispatched twice...
        Event::assertDispatched(OrderShipped::class, 2);

        // Assert an event was not dispatched...
        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}
```

You may pass a closure to the `assertDispatched` or `assertNotDispatched` methods in order to assert that an event was dispatched that passes a given "truth test". If at least one event was dispatched that passes the given truth test then the assertion will be successful:

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

> NOTE After calling `Event::fake()`, no event listeners will be executed. So, if your tests use model factories that rely on events, such as creating a UUID during a model's `creating` event, you should call `Event::fake()` **after** using your factories.

Faking A Subset Of Events

If you only want to fake event listeners for a specific set of events, you may pass them to the `fake` or `fakeFor` method:

```
/**
 * Test order process.
 */
public function test_orders_can_be_processed()
{
    Event::fake([
        OrderCreated::class,
    ]);

    $order = Order::factory()->create();

    Event::assertDispatched(OrderCreated::class);

    // Other events are dispatched as normal...
    $order->update([...]);
}
```

Scoped Event Fakes

If you only want to fake event listeners for a portion of your test, you may use the `fakeFor` method:

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Models\Order;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order process.
     */
    public function test_orders_can_be_processed()
    {
        $order = Event::fakeFor(function () {
            $order = Order::factory()->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });
```

```
        // Events are dispatched as normal and observers will run ...
        $order->update([...]);
    }
}
```

HTTP Fake

The `Http` facade's `fake` method allows you to instruct the HTTP client to return stubbed / dummy responses when requests are made. For more information on faking outgoing HTTP requests, please consult the HTTP Client testing documentation.

Mail Fake

You may use the `Mail` facade's `fake` method to prevent mail from being sent. Typically, sending mail is unrelated to the code you are actually testing. Most likely, it is sufficient to simply assert that Laravel was instructed to send a given mailable.

After calling the `Mail` facade's `fake` method, you may then assert that mailables were instructed to be sent to users and even inspect the data the mailables received:

```
<?php

namespace Tests\Feature;

use App\Mail\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Mail;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Mail::fake();

        // Perform order shipping...

        // Assert that no mailables were sent...
        Mail::assertNothingSent();

        // Assert that a mailable was sent...
        Mail::assertSent(OrderShipped::class);

        // Assert a mailable was sent twice...
        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

If you are queueing mailables for delivery in the background, you should use the `assertQueued` method instead of `assertSent`:

```
Mail::assertQueued(OrderShipped::class);

Mail::assertNotQueued(OrderShipped::class);
```

You may pass a closure to the `assertSent` or `assertNotSent` methods in order to assert that a mailable was sent that passes a given "truth test". If at least one mailable was sent that passes the given truth test then the assertion will be successful:

```
Mail::assertSent(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

When calling the `Mail` facade's assertion methods, the mailable instance accepted by the provided closure exposes helpful methods for examining the recipients of the mailable:

```
Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
```

```
    return $mail->hasTo($user->email) &&
           $mail->hasCc('...') &&
           $mail->hasBcc('...');
});
```

Notification Fake

You may use the `Notification` facade's `fake` method to prevent notifications from being sent. Typically, sending notifications is unrelated to the code you are actually testing. Most likely, it is sufficient to simply assert that Laravel was instructed to send a given notification.

After calling the `Notification` facade's `fake` method, you may then assert that notifications were instructed to be sent to users and even inspect the data the notifications received:

```
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Notification::fake();

        // Perform order shipping...

        // Assert that no notifications were sent...
        Notification::assertNothingSent();

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );
    }
}
```

You may pass a closure to the `assertSentTo` or `assertNotSentTo` methods in order to assert that a notification was sent that passes a given "truth test". If at least one notification was sent that passes the given truth test then the assertion will be successful:

```
Notification::assertSentTo(
    $user,
    function (OrderShipped $notification, $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);
```

On-Demand Notifications

If the code you are testing sends on-demand notifications, you will need to assert that the notification was sent to an `Illuminate\Notifications\AnonymousNotifiable` instance:

```
use Illuminate\Notifications\AnonymousNotifiable;

Notification::assertSentTo(
    new AnonymousNotifiable, OrderShipped::class
);
```

By passing a closure as the third argument to the notification assertion methods, you may determine if an on-demand notification was sent to the correct "route" address:

```
Notification::assertSentTo(
    new AnonymousNotifiable,
    OrderShipped::class,
    function ($notification, $channels, $notifiable) use ($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
```

Queue Fake

You may use the `Queue` facade's `fake` method to prevent queued jobs from being pushed to the queue. Most likely, it is sufficient to simply assert that Laravel was instructed to push a given job to the queue since the queued jobs themselves may be tested in another test class.

After calling the `Queue` facade's `fake` method, you may then assert that the application attempted to push jobs to the queue:

```
<?php

namespace Tests\Feature;

use App\Jobs\AnotherJob;
use App\Jobs\FinalJob;
use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Queue;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Queue::fake();

        // Perform order shipping...

        // Assert that no jobs were pushed...
        Queue::assertNothingPushed();

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was pushed twice...
        Queue::assertPushed(ShipOrder::class, 2);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::class);
    }
}
```

You may pass a closure to the `assertPushed` or `assertNotPushed` methods in order to assert that a job was pushed that passes a given "truth test". If at least one job was pushed that passes the given truth test then the assertion will be successful:

```
Queue::assertPushed(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

Job Chains

The `Queue` facade's `assertPushedWithChain` and `assertPushedWithoutChain` methods may be used to inspect the job chain of a pushed job. The `assertPushedWithChain` method accepts the primary job as its first argument and an array of chained jobs as its second argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Queue;

Queue::assertPushedWithChain(ShipOrder::class, [
    RecordShipment::class,
    UpdateInventory::class
```

```
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Queue::assertPushedWithChain(ShipOrder::class, [
    new RecordShipment,
    new UpdateInventory,
]);
```

You may use the `assertPushedWithoutChain` method to assert that a job was pushed without a chain of jobs:

```
Queue::assertPushedWithoutChain(ShipOrder::class);
```

Storage Fake

The `Storage` facade's `fake` method allows you to easily generate a fake disk that, combined with the file generation utilities of the `Illuminate\Http\UploadedFile` class, greatly simplifies the testing of file uploads. For example:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_albums_can_be_uploaded()
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);

        // Assert one or more files were stored...
        Storage::disk('photos')->assertExists('photo1.jpg');
        Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

        // Assert one or more files were not stored...
        Storage::disk('photos')->assertMissing('missing.jpg');
        Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);
    }
}
```

For more information on testing file uploads, you may consult the [HTTP testing documentation's information on file uploads](#).

> TIP By default, the `fake` method will delete all files in its temporary directory. If you would like to keep these files, you may use the "persistentFake" method instead.

Interacting With Time

When testing, you may occasionally need to modify the time returned by helpers such as `now` or `Illuminate\Support\Carbon::now()`. Thankfully, Laravel's base feature test class includes helpers that allow you to manipulate the current time:

```
public function testTimeCanBeManipulated()
{
    // 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();
}
```

Packages

Laravel Cashier (Stripe)

- Introduction
- Upgrading Cashier
- Installation
  - Database Migrations
- Configuration
  - Billable Model
  - API Keys
  - Currency Configuration
  - Logging
- Customers
  - Retrieving Customers
  - Creating Customers
  - Updating Customers
  - Billing Portal
- Payment Methods
  - Storing Payment Methods
  - Retrieving Payment Methods
  - Determining If A User Has A Payment Method
  - Updating The Default Payment Method
  - Adding Payment Methods
  - Deleting Payment Methods
- Subscriptions
  - Creating Subscriptions
  - Checking Subscription Status
  - Changing Plans
  - Subscription Quantity
  - Multiplan Subscriptions
  - Subscription Taxes
  - Subscription Anchor Date
  - Cancelling Subscriptions
  - Resuming Subscriptions
- Subscription Trials
  - With Payment Method Up Front
  - Without Payment Method Up Front
  - Extending Trials
- Handling Stripe Webhooks
  - Defining Webhook Event Handlers
  - Verifying Webhook Signatures
- Single Charges
  - Simple Charge
  - Charge With Invoice
  - Refunding Charges
- Invoices
  - Retrieving Invoices
  - Generating Invoice PDFs
- Handling Failed Payments
- Strong Customer Authentication (SCA)
  - Payments Requiring Additional Confirmation
  - Off-session Payment Notifications
- Stripe SDK
- Testing

Introduction

Laravel Cashier provides an expressive, fluent interface to Stripe's subscription billing services. It handles

almost all of the boilerplate subscription billing code you are dreading writing. In addition to basic subscription management, Cashier can handle coupons, swapping subscription, subscription "quantities", cancellation grace periods, and even generate invoice PDFs.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review the upgrade guide.

> NOTE To prevent breaking changes, Cashier uses a fixed Stripe API version. Cashier 12 utilizes Stripe API version `2020-03-02`. The Stripe API version will be updated on minor releases in order to make use of new Stripe features and improvements.

Installation

First, install the Cashier package for Stripe using the Composer package manager:

```
composer require laravel/cashier
```

> NOTE To ensure Cashier properly handles all Stripe events, remember to set up Cashier's webhook handling.

Database Migrations

Cashier's service provider registers its own database migration directory, so remember to migrate your database after installing the package. The Cashier migrations will add several columns to your `users` table as well as create a new `subscriptions` table to hold all of your customer's subscriptions:

```
php artisan migrate
```

If you need to overwrite the migrations that ship with Cashier, you can publish them using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

If you would like to prevent Cashier's migrations from running entirely, you may use the `ignoreMigrations` method provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Cashier\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::ignoreMigrations();
}
```

> NOTE Stripe recommends that any column used for storing Stripe identifiers should be case-sensitive. Therefore, you should ensure the column collation for the `stripe_id` column is set to `utf8_bin` when using MySQL. More information regarding this can be found in the Stripe documentation.

Configuration

Billable Model

Before using Cashier, add the `Billable` trait to your billable model definition. Typically, this will be the `App\Models\User` model. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons, and updating payment method information:

```
use Laravel\Cashier\Billable;
```

```
class User extends Authenticatable
{
    use Billable;
}
```

Cashier assumes your billable model will be the `App\Models\User` class that ships with Laravel. If you wish to change this you can specify a different model in your `.env` file:

```
CASHIER_MODEL=App\Models\User
```

> NOTE If you're using a model other than Laravel's supplied `App\Models\User` model, you'll need to publish and alter the Cashier migrations provided to match your alternative model's table name.

API Keys

Next, you should configure your Stripe API keys in your application's `.env` file. You can retrieve your Stripe API keys from the Stripe control panel:

```
STRIPE_KEY=your-stripe-key
STRIPE_SECRET=your-stripe-secret
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by setting the `CASHIER_CURRENCY` environment variable within your application's `.env` file:

```
CASHIER_CURRENCY=eur
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's `NumberFormatter` class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```

> NOTE In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Logging

Cashier allows you to specify the log channel to be used when logging all Stripe related exceptions. You may specify the log channel by defining the `CASHIER_LOGGER` environment variable within your application's `.env` file:

```
CASHIER_LOGGER=stack
```

Customers

Retrieving Customers

You can retrieve a customer by their Stripe ID using the `Cashier::findBillable` method. This method will return an instance of the billable model:

```
use Laravel\Cashier\Cashier;

$user = Cashier::findBillable($stripeId);
```

Creating Customers

Occasionally, you may wish to create a Stripe customer without beginning a subscription. You may accomplish this using the `createAsStripeCustomer` method:

```
$stripeCustomer = $user->createAsStripeCustomer();
```

Once the customer has been created in Stripe, you may begin a subscription at a later date. You may provide an optional `$options` array to pass in any additional customer creation parameters that are supported by the Stripe API:

```
$stripeCustomer = $user->createAsStripeCustomer($options);
```

You may use the `asStripeCustomer` method if you want to return the Stripe customer object for a billable model:

```
$stripeCustomer = $user->asStripeCustomer();
```

The `createOrGetStripeCustomer` method may be used if you would like to retrieve the Stripe customer object for a given billable model but are not sure whether the billable model is already a customer within Stripe. This method will create a new customer in Stripe if one does not already exist:

```
$stripeCustomer = $user->createOrGetStripeCustomer();
```

Updating Customers

Occasionally, you may wish to update the Stripe customer directly with additional information. You may accomplish this using the `updateStripeCustomer` method. This method accepts an array of customer update options supported by the Stripe API:

```
$stripeCustomer = $user->updateStripeCustomer($options);
```

Billing Portal

Stripe offers an easy way to set up a billing portal so that your customer can manage their subscription, payment methods, and view their billing history. You can redirect your users to the billing portal by invoking the `redirectToBillingPortal` method on the billable model from a controller or route:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal();
});
```

By default, when the user is finished managing their subscription, they will be able to return to the `home` route of your application via a link within the Stripe billing portal. You may provide a custom URL that the user should return to by passing the URL as an argument to the `redirectToBillingPortal` method:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal(route('billing'));
});
```

If you would like to generate the URL to the billing portal without generating an HTTP redirect response, you may invoke the `billingPortalUrl` method:

```
$url = $request->user()->billingPortalUrl(route('billing'));
```

Payment Methods

Storing Payment Methods

In order to create subscriptions or perform "one off" charges with Stripe, you will need to store a payment method and retrieve its identifier from Stripe. The approach used to accomplish this differs based on whether you plan to use the payment method for subscriptions or single charges, so we will examine both below.

Payment Methods For Subscriptions

When storing a customer's credit card information for future use by a subscription, the Stripe "Setup Intents" API must be used to securely gather the customer's payment method details. A "Setup Intent" indicates to Stripe the intention to charge a customer's payment method. Cashier's `Billable` trait includes the `createSetupIntent`

method to easily create a new Setup Intent. You should invoke this method from the route or controller that will render the form which gathers your customer's payment method details:

```
return view('update-payment-method', [
    'intent' => $user->createSetupIntent()
]);
```

After you have created the Setup Intent and passed it to the view, you should attach its secret to the element that will gather the payment method. For example, consider this "update payment method" form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->

- `Laravel\Cashier\Exceptions\PaymentActionRequired`: this exception indicates that Stripe requires
extra verification in order to confirm and process a payment.
- `Laravel\Cashier\Exceptions\PaymentFailure`: this exception indicates that a payment failed for
various other reasons, such as being out of available funds.


<a name="strong-customer-authentication"></a>
## Strong Customer Authentication

If your business is based in Europe you will need to abide by the EU's Strong Customer Authentication
(SCA) regulations. These regulations were imposed in September 2019 by the European Union to prevent
payment fraud. Luckily, Stripe and Cashier are prepared for building SCA compliant applications.

> NOTE Before getting started, review [Stripe's guide on PSD2 and SCA](https://stripe.com/guides/strong-
customer-authentication) as well as their [documentation on the new SCA APIs]
(https://stripe.com/docs/strong-customer-authentication).

<a name="payments-requiring-additional-confirmation"></a>
### Payments Requiring Additional Confirmation

SCA regulations often require extra verification in order to confirm and process a payment. When this
happens, Cashier will throw a `Laravel\Cashier\Exceptions\PaymentActionRequired` exception that informs
you that extra verification is needed. More information on how to handle these exceptions be found can
be found in the documentation on [handling failed payments](#handling-failed-payments).

<a name="incomplete-and-past-due-state"></a>
#### Incomplete and Past Due State

When a payment needs additional confirmation, the subscription will remain in an `incomplete` or
`past_due` state as indicated by its `stripe_status` database column. Cashier will automatically
activate the customer's subscription as soon as payment confirmation is complete and your application is
notified by Stripe via webhook of its completion.

For more information on `incomplete` and `past_due` states, please refer to [our additional
documentation on these states](#incomplete-and-past-due-status).

<a name="off-session-payment-notifications"></a>
### Off-Session Payment Notifications

Since SCA regulations require customers to occasionally verify their payment details even while their
subscription is active, Cashier can send a notification to the customer when off-session payment
confirmation is required. For example, this may occur when a subscription is renewing. Cashier's payment
notification can be enabled by setting the `CASHIER_PAYMENT_NOTIFICATION` environment variable to a
notification class. By default, this notification is disabled. Of course, Cashier includes a
notification class you may use for this purpose, but you are free to provide your own notification class
if desired:

    CASHIER_PAYMENT_NOTIFICATION=Laravel\Cashier\Notifications\ConfirmPayment

To ensure that off-session payment confirmation notifications are delivered, verify that [Stripe
webhooks are configured](#handling-stripe-webhooks) for your application and the
`invoice.payment_action_required` webhook is enabled in your Stripe dashboard. In addition, your
`Billable` model should also use Laravel's `Illuminate\Notifications\Notifiable` trait.

> NOTE Notifications will be sent even when customers are manually making a payment that requires
additional confirmation. Unfortunately, there is no way for Stripe to know that the payment was done
manually or "off-session". But, a customer will simply see a "Payment Successful" message if they visit
the payment page after already confirming their payment. The customer will not be allowed to
accidentally confirm the same payment twice and incur an accidental second charge.

<a name="stripe-sdk"></a>
## Stripe SDK

Many of Cashier's objects are wrappers around Stripe SDK objects. If you would like to interact with the
Stripe objects directly, you may conveniently retrieve them using the `asStripe` method:
```

```
    $stripeSubscription = $subscription->asStripeSubscription();

    $stripeSubscription->application_fee_percent = 5;

    $stripeSubscription->save();

You may also use the `updateStripeSubscription` method to update a Stripe subscription directly:

    $subscription->updateStripeSubscription(['application_fee_percent' => 5]);

<a name="testing"></a>
## Testing

When testing an application that uses Cashier, you may mock the actual HTTP requests to the Stripe API;
however, this requires you to partially re-implement Cashier's own behavior. Therefore, we recommend
allowing your tests to hit the actual Stripe API. While this is slower, it provides more confidence that
your application is working as expected and any slow tests may be placed within their own PHPUnit
testing group.

When testing, remember that Cashier itself already has a great test suite, so you should only focus on
testing the subscription and payment flow of your own application and not every underlying Cashier
behavior.

To get started, add the **testing** version of your Stripe secret to your `phpunit.xml` file:

    <env name="STRIPE_SECRET" value="sk_test_<your-key>"/>

Now, whenever you interact with Cashier while testing, it will send actual API requests to your Stripe
testing environment. For convenience, you should pre-fill your Stripe testing account with subscriptions
/ plans that you may use during testing.

> TIP In order to test a variety of billing scenarios, such as credit card denials and failures, you may
use the vast range of [testing card numbers and tokens](https://stripe.com/docs/testing) provided by
Stripe.
```

Packages

Laravel Cashier (Paddle)

- Introduction
- Upgrading Cashier
- Installation
  - Database Migrations
- Configuration
  - Billable Model
  - API Keys
  - Paddle JS
  - Currency Configuration
- Core Concepts
  - Pay Links
  - Inline Checkout
  - User Identification
- Prices
- Customers
  - Customer Defaults
- Subscriptions
  - Creating Subscriptions
  - Checking Subscription Status
  - Subscription Single Charges
  - Updating Payment Information
  - Changing Plans
  - Subscription Quantity
  - Subscription Modifiers
  - Pausing Subscriptions
  - Cancelling Subscriptions
- Subscription Trials
  - With Payment Method Up Front
  - Without Payment Method Up Front
- Handling Paddle Webhooks
  - Defining Webhook Event Handlers
  - Verifying Webhook Signatures
- Single Charges
  - Simple Charge
  - Charging Products
  - Refunding Orders
- Receipts
  - Past & Upcoming Payments
- Handling Failed Payments
- Testing

Introduction

Laravel Cashier Paddle provides an expressive, fluent interface to Paddle's subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading. In addition to basic subscription management, Cashier can handle: coupons, swapping subscription, subscription "quantities", cancellation grace periods, and more.

While working with Cashier we recommend you also review Paddle's user guides and API documentation.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review the upgrade guide.

Installation

First, install the Cashier package for Paddle using the Composer package manager:

```
composer require laravel/cashier-paddle
```

> NOTE To ensure Cashier properly handles all Paddle events, remember to set up Cashier's webhook handling.

Database Migrations

The Cashier service provider registers its own database migration directory, so remember to migrate your database after installing the package. The Cashier migrations will create a new `customers` table. In addition, a new `subscriptions` table will be created to store all of your customer's subscriptions. Finally, a new `receipts` table will be created to store all of your application's receipt information:

```
php artisan migrate
```

If you need to overwrite the migrations that are included with Cashier, you can publish them using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

If you would like to prevent Cashier's migrations from running entirely, you may use the `ignoreMigrations` provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Paddle\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::ignoreMigrations();
}
```

Configuration

Billable Model

Before using Cashier, you must add the `Billable` trait to your user model definition. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons and updating payment method information:

```
use Laravel\Paddle\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

If you have billable entities that are not users, you may also add the trait to those classes:

```
use Illuminate\Database\Eloquent\Model;
use Laravel\Paddle\Billable;

class Team extends Model
{
    use Billable;
}
```

API Keys

Next, you should configure your Paddle keys in your application's `.env` file. You can retrieve your Paddle API keys from the Paddle control panel:

```
PADDLE_VENDOR_ID=your-paddle-vendor-id
PADDLE_VENDOR_AUTH_CODE=your-paddle-vendor-auth-code
PADDLE_PUBLIC_KEY="your-paddle-public-key"
```

Paddle JS

Paddle relies on its own JavaScript library to initiate the Paddle checkout widget. You can load the JavaScript library by placing the `@paddleJS` Blade directive right before your application layout's closing `</head>` tag:

```
<head>
    ...

    @paddleJS
</head>
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by defining a `CASHIER_CURRENCY` environment variable within your application's `.env` file:

```
CASHIER_CURRENCY=EUR
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's `NumberFormatter` class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```

> NOTE In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Core Concepts

Pay Links

Paddle lacks an extensive CRUD API to perform subscription state changes. Therefore, most interactions with Paddle are done through its checkout widget. Before we can display the checkout widget, we must generate a "pay link" using Cashier. A "pay link" will inform the checkout widget of the billing operation we wish to perform:

```
use App\Models\User;
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $request->user()->newSubscription('default', $premium = 34567)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

Cashier includes a `paddle-button` Blade component. We may pass the pay link URL to this component as a "prop". When this button is clicked, Paddle's checkout widget will be displayed:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

By default, this will display a button with the standard Paddle styling. You can remove all Paddle styling by adding the `data-theme="none"` attribute to the component:

```
<x-paddle-button :url="$payLink" class="px-8 py-4" data-theme="none">
    Subscribe
</x-paddle-button>
```

The Paddle checkout widget is asynchronous. Once the user creates or updates a subscription within the widget, Paddle will send your application webhooks so that you may properly update the subscription state in our own database. Therefore, it's important that you properly set up webhooks to accommodate for state changes from Paddle.

For more information on pay links, you may review the Paddle API documentation on pay link generation.

> NOTE After a subscription state change, the delay for receiving the corresponding webhook is typically minimal but you should account for this in your application by considering that your user's subscription might not be immediately available after completing the checkout.

Inline Checkout

If you don't want to make use of Paddle's "overlay" style checkout widget, Paddle also provides the option to display the widget inline. While this approach does not allow you to adjust any of the checkout's HTML fields, it allows you to embed the widget within your application.

To make it easy for you to get started with inline checkout, Cashier includes a `paddle-checkout` Blade component. To get started, you should generate a pay link and pass the pay link to the component's `override` attribute:

```
<x-paddle-checkout :override="$payLink" class="w-full" />
```

To adjust the height of the inline checkout component, you may pass the `height` attribute to the Blade component:

```
<x-paddle-checkout :override="$payLink" class="w-full" height="500" />
```

Inline Checkout Without Pay Links

Alternatively, you may customize the widget with custom options instead of using a pay link:

```
$options = [
    'product' => $productId,
    'title' => 'Product Title',
];

<x-paddle-checkout :options="$options" class="w-full" />
```

Please consult Paddle's guide on Inline Checkout as well as their parameter reference for further details on the inline checkout's available options.

> NOTE If you would like to also use the `passthrough` option when specifying custom options, you should provide a key / value array as its value. Cashier will automatically handle converting the array to a JSON string. In addition, the `customer_id` passthrough option is reserved for internal Cashier usage.

User Identification

In contrast to Stripe, Paddle users are unique across all of Paddle, not unique per Paddle account. Because of this, Paddle's API's do not currently provide a method to update a user's details such as their email address. When generating pay links, Paddle identifies users using the `customer_email` parameter. When creating a subscription, Paddle will try to match the user provided email to an existing Paddle user.

In light of this behavior, there are some important things to keep in mind when using Cashier and Paddle. First, you should be aware that even though subscriptions in Cashier are tied to the same application user, **they could be tied to different users within Paddle's internal systems**. Secondly, each subscription has its own connected payment method information and could also have different email addresses within Paddle's internal systems (depending on which email was assigned to the user when the subscription was created).

Therefore, when displaying subscriptions you should always inform the user which email address or payment method information is connected to the subscription on a per-subscription basis. Retrieving this information can be done with the following methods provided by the `Laravel\Paddle\Subscription` model:

```
$subscription = $user->subscription('default');

$subscription->paddleEmail();
$subscription->paymentMethod();
$subscription->cardBrand();
$subscription->cardLastFour();
$subscription->cardExpirationDate();
```

There is currently no way to modify a user's email address through the Paddle API. When a user wants to update their email address within Paddle, the only way for them to do so is to contact Paddle customer support. When communicating with Paddle, they need to provide the `paddleEmail` value of the subscription to assist Paddle in updating the correct user.

Prices

Paddle allows you to customize prices per currency, essentially allowing you to configure different prices for different countries. Cashier Paddle allows you to retrieve all of the prices for a given product using the `productPrices` method. This method accepts the product IDs of the products you wish to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456]);
```

The currency will be determined based on the IP address of the request; however, you may optionally provide a specific country to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456], ['customer_country' => 'BE']);
```

After retrieving the prices you may display them however you wish:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->gross() }}</li>
    @endforeach
</ul>
```

You may also display the net price (excludes tax) and display the tax amount separately:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->net() }} (+ {{ $price->price()->tax() }}
tax)</li>
    @endforeach
</ul>
```

If you retrieved prices for subscription plans you can display their initial and recurring price separately:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - Initial: {{ $price->initialPrice()->gross() }} - Recurring: {{
$price->recurringPrice()->gross() }}</li>
    @endforeach
</ul>
```

For more information, check Paddle's API documentation on prices.

Customers

If a user is already a customer and you would like to display the prices that apply to that customer, you may do so by retrieving the prices directly from the customer instance:

```
use App\Models\User;

$prices = User::find(1)->productPrices([123, 456]);
```

Internally, Cashier will use the user's `paddleCountry` method to retrieve the prices in their currency. So, for example, a user living in the United States will see prices in USD while a user in Belgium will see prices in EUR. If no matching currency can be found the default currency of the product will be used. You can customize

all prices of a product or subscription plan in the Paddle control panel.

Coupons

You may also choose to display prices after a coupon reduction. When calling the `productPrices` method, coupons may be passed as a comma delimited string:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456], [
    'coupons' => 'SUMMERSALE,20PERCENTOFF'
]);
```

Then, display the calculated prices using the `price` method:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->gross() }}</li>
    @endforeach
</ul>
```

You may display the original listed prices (without coupon discounts) using the `listPrice` method:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->listPrice()->gross() }}</li>
    @endforeach
</ul>
```

> NOTE When using the prices API, Paddle only allows to apply coupons to one-time purchase products and not to subscription plans.

Customers

Customer Defaults

Cashier allows you to define some useful defaults for your customers when creating pay links. Setting these defaults allow you to pre-fill a customer's email address, country, and postal code so that they can immediately move on to the payment portion of the checkout widget. You can set these defaults by overriding the following methods on your billable model:

```
/**
 * Get the customer's email address to associate with Paddle.
 *
 * @return string|null
 */
public function paddleEmail()
{
    return $this->email;
}

/**
 * Get the customer's country to associate with Paddle.
 *
 * This needs to be a 2 letter code. See the link below for supported countries.
 *
 * @return string|null
 * @link https://developer.paddle.com/reference/platform-parameters/supported-countries
 */
public function paddleCountry()
{
    //
}

/**
 * Get the customer's postal code to associate with Paddle.
 *
 * See the link below for countries which require this.
 *
 * @return string|null
 * @link https://developer.paddle.com/reference/platform-parameters/supported-countries#countries-
requiring-postcode
```

```
 */
public function paddlePostcode()
{
    //
}
```

These defaults will be used for every action in Cashier that generates a pay link.

Subscriptions

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\Models\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription pay link:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $user->newSubscription('default', $premium = 12345)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

The first argument passed to the `newSubscription` method should be the name of the subscription. If your application only offers a single subscription, you might call this `default` or `primary`. The second argument is the specific plan the user is subscribing to. This value should correspond to the plan's identifier in Paddle. The `returnTo` method accepts a URL that your user will be redirected to after they successfully complete the checkout.

The `create` method will create a pay link which you can use to generate a payment button. The payment button can be generated using the `paddle-button` Blade component that is included with Cashier Paddle:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

After the user has finished their checkout, a `subscription_created` webhook will be dispatched from Paddle. Cashier will receive this webhook and setup the subscription for your customer. In order to make sure all webhooks are properly received and handled by your application, ensure you have properly setup webhook handling.

Additional Details

If you would like to specify additional customer or subscription details, you may do so by passing them as an array of key / value pairs to the `create` method. To learn more about the additional fields supported by Paddle, check out Paddle's documentation on generating pay links:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->create([
        'vat_number' => $vatNumber,
    ]);
```

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->withCoupon('code')
    ->create();
```

Metadata

You can also pass an array of metadata using the `withMetadata` method:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->withMetadata(['key' => 'value'])
    ->create();
```

> NOTE When providing metadata, please avoid using `subscription_name` as a metadata key. This key is reserved for internal use by Cashier.

Checking Subscription Status

Once a user is subscribed to your application, you may check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the user has an active subscription, even if the subscription is currently within its trial period:

```
if ($user->subscribed('default')) {
    //
}
```

The `subscribed` method also makes a great candidate for a route middleware, allowing you to filter access to routes and controllers based on the user's subscription status:

```
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureUserIsSubscribed
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->user() && ! $request->user()->subscribed('default')) {
            // This user is not a paying customer...
            return redirect('billing');
        }

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

If you would like to determine if a user is still within their trial period, you may use the `onTrial` method. This method can be useful for determining if you should display a warning to the user that they are still on their trial period:

```
if ($user->subscription('default')->onTrial()) {
    //
}
```

The `subscribedToPlan` method may be used to determine if the user is subscribed to a given plan based on a given Paddle plan ID. In this example, we will determine if the user's `default` subscription is actively subscribed to the monthly plan:

```
if ($user->subscribedToPlan($monthly = 12345, 'default')) {
    //
}
```

By passing an array to the `subscribedToPlan` method, you may determine if the user's `default` subscription is actively subscribed to the monthly or the yearly plan:

```
if ($user->subscribedToPlan([$monthly = 12345, $yearly = 54321], 'default')) {
    //
}
```

The `recurring` method may be used to determine if the user is currently subscribed and is no longer within their trial period:

```
if ($user->subscription('default')->recurring()) {
    //
}
```

Cancelled Subscription Status

To determine if the user was once an active subscriber but has cancelled their subscription, you may use the `cancelled` method:

```
if ($user->subscription('default')->cancelled()) {
    //
}
```

You may also determine if a user has cancelled their subscription, but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. Note that the `subscribed` method still returns `true` during this time:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

To determine if the user has cancelled their subscription and is no longer within their "grace period", you may use the `ended` method:

```
if ($user->subscription('default')->ended()) {
    //
}
```

Past Due Status

If a payment fails for a subscription, it will be marked as `past_due`. When your subscription is in this state it will not be active until the customer has updated their payment information. You may determine if a subscription is past due using the `pastDue` method on the subscription instance:

```
if ($user->subscription('default')->pastDue()) {
    //
}
```

When a subscription is past due, you should instruct the user to update their payment information. You may configure how past due subscriptions are handled in your Paddle subscription settings.

If you would like subscriptions to still be considered active when they are `past_due`, you may use the `keepPastDueSubscriptionsActive` method provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Paddle\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::keepPastDueSubscriptionsActive();
}
```

> NOTE When a subscription is in a `past_due` state it cannot be changed until payment information has been updated. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in a `past_due` state.

Subscription Scopes

Most subscription states are also available as query scopes so that you may easily query your database for

subscriptions that are in a given state:

```
// Get all active subscriptions...
$subscriptions = Subscription::query()->active()->get();

// Get all of the cancelled subscriptions for a user...
$subscriptions = $user->subscriptions()->cancelled()->get();
```

A complete list of available scopes is available below:

```
Subscription::query()->active();
Subscription::query()->onTrial();
Subscription::query()->notOnTrial();
Subscription::query()->pastDue();
Subscription::query()->recurring();
Subscription::query()->ended();
Subscription::query()->paused();
Subscription::query()->notPaused();
Subscription::query()->onPausedGracePeriod();
Subscription::query()->notOnPausedGracePeriod();
Subscription::query()->cancelled();
Subscription::query()->notCancelled();
Subscription::query()->onGracePeriod();
Subscription::query()->notOnGracePeriod();
```

Subscription Single Charges

Subscription single charges allow you to charge subscribers with a one-time charge on top of their subscriptions:

```
$response = $user->subscription('default')->charge(12.99, 'Support Add-on');
```

In contrast to single charges, this method will immediately charge the customer's stored payment method for the subscription. The charge amount should always be defined in the currency of the subscription.

Updating Payment Information

Paddle always saves a payment method per subscription. If you want to update the default payment method for a subscription, you should first generate a subscription "update URL" using the `updateUrl` method on the subscription model:

```
use App\Models\User;

$user = User::find(1);

$updateUrl = $user->subscription('default')->updateUrl();
```

Then, you may use the generated URL in combination with Cashier's provided `paddle-button` Blade component to allow the user to initiate the Paddle widget and update their payment information:

```
<x-paddle-button :url="$updateUrl" class="px-8 py-4">
    Update Card
</x-paddle-button>
```

When a user has finished updating their information, a `subscription_updated` webhook will be dispatched by Paddle and the subscription details will be updated in your application's database.

Changing Plans

After a user has subscribed to your application, they may occasionally want to change to a new subscription plan. To update the subscription plan for a user, you should pass the Paddle plan's identifier to the subscription's `swap` method:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->swap($premium = 34567);
```

If the user is on a trial, the trial period will be maintained. Also, if a "quantity" exists for the subscription, that quantity will also be maintained.

If you would like to swap plans and cancel any trial period the user is currently on, you may use the `skipTrial` method:

```
$user->subscription('default')
        ->skipTrial()
        ->swap($premium = 34567);
```

If you would like to swap plans and immediately invoice the user instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = User::find(1);

$user->subscription('default')->swapAndInvoice($premium = 34567);
```

Prorations

By default, Paddle prorates charges when swapping between plans. The `noProrate` method may be used to update the subscription's without prorating the charges:

```
$user->subscription('default')->noProrate()->swap($premium = 34567);
```

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, a project management application might charge $10 per month per project. To easily increment or decrement your subscription's quantity, use the `incrementQuantity` and `decrementQuantity` methods:

```
$user = User::find(1);

$user->subscription('default')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('default')->incrementQuantity(5);

$user->subscription('default')->decrementQuantity();

// Subtract five from the subscription's current quantity...
$user->subscription('default')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('default')->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without prorating the charges:

```
$user->subscription('default')->noProrate()->updateQuantity(10);
```

Subscription Modifiers

Subscription modifiers allow you to implement metered billing or extend subscriptions with add-ons.

For example, you might want to offer a "Premium Support" add-on with your standard subscription. You can create this modifier like so:

```
$modifier = $user->subscription->newModifier(12.99)->create();
```

The example above will add a $12.99 add-on to the subscription. By default, this charge will recur on every interval you have configured for the subscription. If you would like, you can add a readable description to the modifier using the modifier's `description` method:

```
$modifier = $user->subscription->newModifier(12.99)
    ->description('Premium Support')
    ->create();
```

To illustrate how to implement metered billing using modifiers, imagine your application charges per SMS

message sent by the user. First, you should create a $0 plan in your Paddle dashboard. Once the user has been subscribed to this plan, you can add modifiers representing each individual charge to the subscription:

```
$modifier = $user->subscription->newModifier(0.99)
    ->description('New text message')
    ->oneTime()
    ->create();
```

As you can see, we invoked the `oneTime` method when creating this modifier. This method will ensure the modifier is only charged once and does not recur every billing interval.

Retrieving Modifiers

You may retrieve a list of all modifiers for a subscription via the `modifiers` method:

```
$modifiers = $user->subscription->modifiers();

foreach ($modifiers as $modifier) {
    $modifier->amount(); // $0.99
    $modifier->description; // "New text message."
}
```

Deleting Modifiers

Modifiers may be deleted by invoking the `delete` method on a `Laravel\Paddle\Modifier` instance:

```
$modifier->delete();
```

Pausing Subscriptions

To pause a subscription, call the `pause` method on the user's subscription:

```
$user->subscription('default')->pause();
```

When a subscription is paused, Cashier will automatically set the `paused_from` column in your database. This column is used to know when the `paused` method should begin returning `true`. For example, if a customer pauses a subscription on March 1st, but the subscription was not scheduled to recur until March 5th, the `paused` method will continue to return `false` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has paused their subscription but are still on their "grace period" using the `onPausedGracePeriod` method:

```
if ($user->subscription('default')->onPausedGracePeriod()) {
    //
}
```

To resume a paused a subscription, you may call the `unpause` method on the user's subscription:

```
$user->subscription('default')->unpause();
```

> NOTE A subscription cannot be modified while it is paused. If you want to swap to a different plan or update quantities you must resume the subscription first.

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription('default')->cancel();
```

When a subscription is cancelled, Cashier will automatically set the `ends_at` column in your database. This column is used to know when the `subscribed` method should begin returning `false`. For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has cancelled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

If you wish to cancel a subscription immediately, you may call the `cancelNow` method on the user's subscription:

```
$user->subscription('default')->cancelNow();
```

> NOTE Paddle's subscriptions cannot be resumed after cancellation. If your customer wishes to resume their subscription, they will have to subscribe to a new subscription.

Subscription Trials

With Payment Method Up Front

> NOTE While trialing and collecting payment method details up front, Paddle prevents any subscription changes such as swapping plans or updating quantities. If you want to allow a customer to swap plans during a trial the subscription must be cancelled and recreated.

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should use the `trialDays` method when creating your subscription pay links:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $request->user()->newSubscription('default', $monthly = 12345)
                ->returnTo(route('home'))
                ->trialDays(10)
                ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

This method will set the trial period ending date on the subscription record within your application's database, as well as instruct Paddle to not begin billing the customer until after this date.

> NOTE If the customer's subscription is not cancelled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

You may determine if the user is within their trial period using either the `onTrial` method of the user instance or the `onTrial` method of the subscription instance. The two examples below are equivalent:

```
if ($user->onTrial('default')) {
    //
}

if ($user->subscription('default')->onTrial()) {
    //
}
```

Defining Trial Days In Paddle / Cashier

You may choose to define how many trial days your plan's receive in the Paddle dashboard or always pass them explicitly using Cashier. If you choose to define your plan's trial days in Paddle you should be aware that new subscriptions, including new subscriptions for a customer that had a subscription in the past, will always receive a trial period unless you explicitly call the `trialDays(0)` method.

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the customer record attached to your user to your desired trial ending date. This is typically done during user registration:

```
use App\Models\User;

$user = User::create([
    // ...
]);

$user->createAsCustomer([
    'trial_ends_at' => now()->addDays(10)
]);
```

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the `User` instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $user->newSubscription('default', $monthly = 12345)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

To retrieve the user's trial ending date, you may use the `trialEndsAt` method. This method will return a Carbon date instance if a user is on a trial or `null` if they aren't. You may also pass an optional subscription name parameter if you would like to get the trial ending date for a specific subscription other than the default one:

```
if ($user->onTrial()) {
    $trialEndsAt = $user->trialEndsAt('main');
}
```

You may use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not created an actual subscription yet:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

> NOTE There is no way to extend or modify a trial period on a Paddle subscription after it has been created.

Handling Paddle Webhooks

Paddle can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is registered by the Cashier service provider. This controller will handle all incoming webhook requests.

By default, this controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Paddle subscription settings), subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any Paddle webhook event you like.

To ensure your application can handle Paddle webhooks, be sure to configure the webhook URL in the Paddle control panel. By default, Cashier's webhook controller responds to the `/paddle/webhook` URL path. The full list of all webhooks you should enable in the Paddle control panel are:

- Subscription Created
- Subscription Updated
- Subscription Cancelled
- Payment Succeeded
- Subscription Payment Succeeded

NOTE Make sure you protect incoming requests with Cashier's included webhook signature verification middleware.

Webhooks & CSRF Protection

Since Paddle webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your `App\Http\Middleware\VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'paddle/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellation on failed charges and other common Paddle webhooks, but if you have additional webhook events you would like to handle, you should extend Cashier's `WebhookController`.

Your controller's method names should correspond to Cashier's controller method conventions. Specifically, methods should be prefixed with `handle` and the "camel case" name of the webhook you wish to handle. For example, if you wish to handle the `payment_succeeded` webhook, you should add a `handlePaymentSucceeded` method to the controller:

```
<?php

namespace App\Http\Controllers;

use Laravel\Paddle\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle payment succeeded.
     *
     * @param  array  $payload
     * @return void
     */
    public function handlePaymentSucceeded($payload)
    {
        // Handle the event...
    }
}
```

Next, define a route to your Cashier webhook controller within your application's `routes/web.php` file. This will overwrite the default route registered by Cashier's service provider:

```
use App\Http\Controllers\WebhookController;

Route::post('/paddle/webhook', WebhookController::class);
```

Cashier emits a `Laravel\Paddle\Events\WebhookReceived` event when a webhook is received and a `Laravel\Paddle\Events\WebhookHandled` event when a webhook was handled. Both events contain the full payload of the Paddle webhook.

Cashier also emit events dedicated to the type of the received webhook. In addition to the full payload from Paddle, they also contain the relevant models that were used to process the webhook such as the billable model, the subscription, or the receipt:

- `Laravel\Paddle\Events\PaymentSucceeded`
- `Laravel\Paddle\Events\SubscriptionPaymentSucceeded`
- `Laravel\Paddle\Events\SubscriptionCreated`
- `Laravel\Paddle\Events\SubscriptionUpdated`
- `Laravel\Paddle\Events\SubscriptionCancelled`

You can also override the default, built-in webhook route by defining the `CASHIER_WEBHOOK` environment variable in your application's `.env` file. This value should be the full URL to your webhook route and needs to match the URL set in your Paddle control panel:

```
CASHIER_WEBHOOK=https://example.com/my-paddle-webhook-url
```

Verifying Webhook Signatures

To secure your webhooks, you may use Paddle's webhook signatures. For convenience, Cashier automatically includes a middleware which validates that the incoming Paddle webhook request is valid.

To enable webhook verification, ensure that the `PADDLE_PUBLIC_KEY` environment variable is defined in your application's `.env` file. The public key may be retrieved from your Paddle account dashboard.

Single Charges

Simple Charge

If you would like to make a one-time charge against a customer, you may use the `charge` method on a billable model instance to generate a pay link for the charge. The `charge` method accepts the charge amount (float) as its first argument and a charge description as its second argument:

```
use Illuminate\Http\Request;

Route::get('/store', function (Request $request) {
    return view('store', [
        'payLink' => $user->charge(12.99, 'Action Figure')
    ]);
});
```

After generating the pay link, you may use Cashier's provided `paddle-button` Blade component to allow the user to initiate the Paddle widget and complete the charge:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Buy
</x-paddle-button>
```

The `charge` method accepts an array as its third argument, allowing you to pass any options you wish to the underlying Paddle pay link creation. Please consult the Paddle documentation to learn more about the options available to you when creating charges:

```
$payLink = $user->charge(12.99, 'Action Figure', [
    'custom_option' => $value,
]);
```

Charges happen in the currency specified in the `cashier.currency` configuration option. By default, this is set to USD. You may override the default currency by defining the `CASHIER_CURRENCY` environment variable in your application's `.env` file:

```
CASHIER_CURRENCY=EUR
```

You can also override prices per currency using Paddle's dynamic pricing matching system. To do so, pass an array of prices instead of a fixed amount:

```
$payLink = $user->charge([
    'USD:19.99',
    'EUR:15.99',
], 'Action Figure');
```

Charging Products

If you would like to make a one-time charge against a specific product configured within Paddle, you may use the `chargeProduct` method on a billable model instance to generate a pay link:

```
use Illuminate\Http\Request;

Route::get('/store', function (Request $request) {
    return view('store', [
        'payLink' => $request->user()->chargeProduct($productId = 123)
    ]);
});
```

Then, you may provide the pay link to the `paddle-button` component to allow the user to initialize the Paddle widget:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Buy
</x-paddle-button>
```

The `chargeProduct` method accepts an array as its second argument, allowing you to pass any options you wish to the underlying Paddle pay link creation. Please consult the Paddle documentation regarding the options that are available to you when creating charges:

```
$payLink = $user->chargeProduct($productId, [
    'custom_option' => $value,
]);
```

Refunding Orders

If you need to refund a Paddle order, you may use the `refund` method. This method accepts the Paddle order ID as its first argument. You may retrieve the receipts for a given billable model using the `receipts` method:

```
use App\Models\User;

$user = User::find(1);

$receipt = $user->receipts()->first();

$refundRequestId = $user->refund($receipt->order_id);
```

You may optionally specify a specific amount to refund as well as a reason for the refund:

```
$receipt = $user->receipts()->first();

$refundRequestId = $user->refund(
    $receipt->order_id, 5.00, 'Unused product time'
);
```

TIP You can use the `$refundRequestId` as a reference for the refund when contacting Paddle support.

Receipts

You may easily retrieve an array of a billable model's receipts using the `receipts` method:

```
use App\Models\User;

$user = User::find(1);

$receipts = $user->receipts();
```

When listing the receipts for the customer, you may use the receipt instance's methods to display the relevant receipt information. For example, you may wish to list every receipt in a table, allowing the user to easily download any of the receipts:

```
<table>
    @foreach ($receipts as $receipt)
        <tr>
            <td>{{ $receipt->paid_at->toFormattedDateString() }}</td>
            <td>{{ $receipt->amount() }}</td>
            <td><a href="{{ $receipt->receipt_url }}" target="_blank">Download</a></td>
        </tr>
    @endforeach
</table>
```

Past & Upcoming Payments

You may use the `lastPayment` and `nextPayment` methods to retrieve and display a customer's past or upcoming payments for recurring subscriptions:

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

```
$user = User::find(1);

$subscription = $user->subscription('default');

$lastPayment = $subscription->lastPayment();
$nextPayment = $subscription->nextPayment();
```

Both of these methods will return an instance of `Laravel\Paddle\Payment`; however, `nextPayment` will return `null` when the billing cycle has ended (such as when a subscription has been cancelled):

```
Next payment: {{ $nextPayment->amount() }} due on {{ $nextPayment->date()->format('d/m/Y') }}
```

Handling Failed Payments

Subscription payments fail for various reasons, such as expired cards or a card having insufficient funds. When this happens, we recommend that you let Paddle handle payment failures for you. Specifically, you may setup Paddle's automatic billing emails in your Paddle dashboard.

Alternatively, you can perform more precise customization by catching the `subscription_payment_failed` webhook and enabling the "Subscription Payment Failed" option in the Webhook settings of your Paddle dashboard:

```
<?php

namespace App\Http\Controllers;

use Laravel\Paddle\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle subscription payment failed.
     *
     * @param  array  $payload
     * @return void
     */
    public function handleSubscriptionPaymentFailed($payload)
    {
        // Handle the failed subscription payment...
    }
}
```

Testing

Paddle currently lacks a proper CRUD API so you will need to manually test your billing flow. Paddle also lacks a sandboxed developer environment so any card charges you make are live charges. In order to work around this, we recommend you use coupons with a 100% discount or free products during testing.

Packages

Laravel Dusk

- Introduction
- Installation
  - Managing ChromeDriver Installations
  - Using Other Browsers
- Getting Started
  - Generating Tests
  - Database Migrations
  - Running Tests
  - Environment Handling
- Browser Basics
  - Creating Browsers
  - Navigation
  - Resizing Browser Windows
  - Browser Macros
  - Authentication
  - Cookies
  - Executing JavaScript
  - Taking A Screenshot
  - Storing Console Output To Disk
  - Storing Page Source To Disk
- Interacting With Elements
  - Dusk Selectors
  - Text, Values, & Attributes
  - Interacting With Forms
  - Attaching Files
  - Pressing Buttons
  - Clicking Links
  - Using The Keyboard
  - Using The Mouse
  - JavaScript Dialogs
  - Scoping Selectors
  - Waiting For Elements
  - Scrolling An Element Into View
- Available Assertions
- Pages
  - Generating Pages
  - Configuring Pages
  - Navigating To Pages
  - Shorthand Selectors
  - Page Methods
- Components
  - Generating Components
  - Using Components
- Continuous Integration
  - Heroku CI
  - Travis CI
  - GitHub Actions

Introduction

Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your local computer. Instead, Dusk uses a standalone ChromeDriver installation. However, you are free to utilize any other Selenium compatible driver you wish.

Installation

To get started, you should add the `laravel/dusk` Composer dependency to your project:

```
composer require --dev laravel/dusk
```

> NOTE If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.

After installing the Dusk package, execute the `dusk:install` Artisan command. The `dusk:install` command will create a `tests/Browser` directory and an example Dusk test:

```
php artisan dusk:install
```

Next, set the `APP_URL` environment variable in your application's `.env` file. This value should match the URL you use to access your application in a browser.

> TIP If you are using Laravel Sail to manage your local development environment, please also consult the Sail documentation on configuring and running Dusk tests.

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is included with Laravel Dusk, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 86

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all

# Install the version of ChromeDriver that matches the detected version of Chrome / Chromium for your
OS...
php artisan dusk:chrome-driver --detect
```

> NOTE Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone ChromeDriver installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the ChromeDriver:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
/**
```

```
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Database Migrations

Most of the tests you write will interact with pages that retrieve data from your application's database; however, your Dusk tests should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which will not be applicable or available across HTTP requests. Instead, use the `DatabaseMigrations` trait, which re-migrates the database for each test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

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

> NOTE SQLite in-memory databases may not be used when executing Dusk tests. Since the browser executes within its own process, it will not be able to access the in-memory databases of other processes.

Running Tests

To run your browser tests, execute the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, such as allowing you to only run the tests for a given group:

```
php artisan dusk --group=foo
```

> TIP If you are using Laravel Sail to manage your local development environment, please consult the Sail documentation on configuring and running Dusk tests.

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular

system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class to reflect the correct port:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a `.env.dusk.local` file.

When running tests, Dusk will back-up your `.env` file and rename your Dusk environment to `.env`. Once the tests have completed, your `.env` file will be restored.

Browser Basics

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, you may call the `browse` method from within your Dusk test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function test_basic_example()
    {
        $user = User::factory()->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
```

```
            $browser->visit('/login')
                    ->type('email', $user->email)
                    ->type('password', 'password')
                    ->press('Login')
                    ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a closure. A browser instance will automatically be passed to this closure by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, simply add more browser arguments to the signature of the closure given to the `browse` method:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home')
          ->waitForText('Message');

    $second->loginAs(User::find(2))
           ->visit('/home')
           ->waitForText('Message')
           ->type('message', 'Hey Taylor')
           ->press('Send');

    $first->waitForText('Hey Taylor')
          ->assertSee('Jeffrey Way');
});
```

Navigation

The `visit` method may be used to navigate to a given URI within your application:

```
$browser->visit('/login');
```

You may use the `visitRoute` method to navigate to a named route:

```
$browser->visitRoute('login');
```

You may navigate "back" and "forward" using the `back` and `forward` methods:

```
$browser->back();

$browser->forward();
```

You may use the `refresh` method to refresh the page:

```
$browser->refresh();
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

The `fitContent` method will resize the browser window to match the size of its content:

```
$browser->fitContent();
```

When a test fails, Dusk will automatically resize the browser to fit the content prior to taking a screenshot. You

may disable this feature by calling the `disableFitOnFailure` method within your test:

```
$browser->disableFitOnFailure();
```

You may use the `move` method to move the browser window to a different position on your screen:

```
$browser->move($x = 100, $y = 100);
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     *
     * @return void
     */
    public function boot()
    {
        Browser::macro('scrollToElement', function ($element = null) {
            $this->script("$('html, body').animate({ scrollTop: $('$element').offset().top }, 0);");

            return $this;
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Browser` instance:

```
$this->browse(function ($browser) use ($user) {
    $browser->visit('/pay')
            ->scrollToElement('#credit-card-details')
            ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with your application's login screen during every test. The `loginAs` method accepts a primary key associated with your authenticatable model or an authenticatable model instance:

```
use App\Models\User;

$this->browse(function ($browser) {
    $browser->loginAs(User::find(1))
          ->visit('/home');
});
```

NOTE After using the `loginAs` method, the user session will be maintained for all tests within the file.

Cookies

You may use the `cookie` method to get or set an encrypted cookie's value. By default, all of the cookies created by Laravel are encrypted:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

You may use the `plainCookie` method to get or set an unencrypted cookie's value:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

You may use the `deleteCookie` method to delete the given cookie:

```
$browser->deleteCookie('name');
```

Executing JavaScript

You may use the `script` method to execute arbitrary JavaScript statements within the browser:

```
$output = $browser->script('document.documentElement.scrollTop = 0');

$output = $browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);
```

Taking A Screenshot

You may use the `screenshot` method to take a screenshot and store it with the given filename. All screenshots will be stored within the `tests/Browser/screenshots` directory:

```
$browser->screenshot('filename');
```

Storing Console Output To Disk

You may use the `storeConsoleLog` method to write the current browser's console output to disk with the given filename. Console output will be stored within the `tests/Browser/console` directory:

```
$browser->storeConsoleLog('filename');
```

Storing Page Source To Disk

You may use the `storeSource` method to write the current page's source to disk with the given filename. The page source will be stored within the `tests/Browser/source` directory:

```
$browser->storeSource('filename');
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, when interacting with a Dusk browser, prefix the selector with `@` to manipulate the attached element within your test:

```
// HTML...

<button dusk="login-button">Login</button>

// Test...
```

```
$browser->click('@login-button');
```

Text, Values, & Attributes

Retrieving & Setting Values

Dusk provides several methods for interacting with the current display text, value, and attributes of elements on the page. For example, to get the "value" of an element that matches a given CSS or Dusk selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

You may use the `inputValue` method to get the "value" of an input element that has a given field name:

```
$value = $browser->inputValue('field');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve the value of an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Interacting With Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an `input` or `textarea` field with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
        ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

You can instruct Dusk to type slowly using the `typeSlowly` method. By default, Dusk will pause for 100 milliseconds between key presses. To customize the amount of time between key presses, you may pass the appropriate number of milliseconds as the third argument to the method:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');

$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

You may use the `appendSlowly` method to append text slowly:

```
$browser->type('tags', 'foo')
        ->appendSlowly('tags', ', bar, baz');
```

Dropdowns

To select a value available on a `select` element, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second argument:

```
$browser->select('size');
```

Checkboxes

To "check" a checkbox input, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');
```

The `uncheck` method may be used to "uncheck" a checkbox input:

```
$browser->uncheck('terms');
```

Radio Buttons

To "select" a `radio` input option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `radio` input with matching `name` and `value` attributes:

```
$browser->radio('size', 'large');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `file` input with a matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```

> NOTE The attach function requires the `Zip` PHP extension to be installed and enabled on your server.

Pressing Buttons

The `press` method may be used to click a button element on the page. The first argument given to the `press` method may be either the display text of the button or a CSS / Dusk selector:

```
$browser->press('Login');
```

When submitting forms, many application's disable the form's submission button after it are pressed and then re-enable the button when the form submission's HTTP request is complete. To press a button and wait for the button to be re-enabled, you may use the `pressAndWaitFor` method:

```
// Press the button and wait a maximum of 5 seconds for it to be enabled...
$browser->pressAndWaitFor('Save');

// Press the button and wait a maximum of 1 second for it to be enabled...
$browser->pressAndWaitFor('Save', 1);
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

You may use the `seeLink` method to determine if a link with the given display text is visible on the page:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```

> NOTE These methods interact with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Using The Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may instruct Dusk to hold modifier keys while entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `swift` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Another valuable use case for the `keys` method is sending a "keyboard shortcut" combination to the primary CSS selector for your application:

```
$browser->keys('.app', ['{command}', 'j']);
```

> TIP All modifier keys such as `{command}` are wrapped in `{}` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be found on GitHub.

Using The Mouse

Clicking On Elements

The `click` method may be used to click on an element matching the given CSS or Dusk selector:

```
$browser->click('.selector');
```

The `clickAtXPath` method may be used to click on an element matching the given XPath expression:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

The `clickAtPoint` method may be used to click on the topmost element at a given pair of coordinates relative to the viewable area of the browser:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

The `doubleClick` method may be used to simulate the double click of a mouse:

```
$browser->doubleClick();
```

The `rightClick` method may be used to simulate the right click of a mouse:

```
$browser->rightClick();
```

```
$browser->rightClick('.selector');
```

The `clickAndHold` method may be used to simulate a mouse button being clicked and held down. A subsequent call to the `releaseMouse` method will undo this behavior and release the mouse button:

```
$browser->clickAndHold()
        ->pause(1000)
        ->releaseMouse();
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given CSS or Dusk selector:

```
$browser->mouseover('.selector');
```

Drag & Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', $pixels = 10);
$browser->dragRight('.selector', $pixels = 10);
$browser->dragUp('.selector', $pixels = 10);
$browser->dragDown('.selector', $pixels = 10);
```

Finally, you may drag an element by a given offset:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs. For example, you may use the `waitForDialog` method to wait for a JavaScript dialog to appear. This method accepts an optional argument indicating how many seconds to wait for the dialog to appear:

```
$browser->waitForDialog($seconds = null);
```

The `assertDialogOpened` method may be used to assert that a dialog has been displayed and contains the given message:

```
$browser->assertDialogOpened('Dialog message');
```

If the JavaScript dialog contains a prompt, you may use the `typeInDialog` method to type a value into the prompt:

```
$browser->typeInDialog('Hello World');
```

To close an open JavaScript dialog by clicking the "OK" button, you may invoke the `acceptDialog` method:

```
$browser->acceptDialog();
```

To close an open JavaScript dialog by clicking the "Cancel" button, you may invoke the `dismissDialog` method:

```
$browser->dismissDialog();
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to accomplish this. All operations performed within the closure given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

You may occasionally need to execute assertions outside of the current scope. You may use the `elsewhere` method to accomplish this:

```
 $browser->with('.table', function ($table) {
    // Current scope is `body .table`...

    $browser->elsewhere('.page-title', function ($title) {
```

```
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });
});
```

Waiting For Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to be visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you just need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

Waiting For Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS or Dusk selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

You may also wait until the element matching the given selector contains the given text:

```
// Wait a maximum of five seconds for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World');

// Wait a maximum of one second for the selector to contain the given text...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

You may also wait until the element matching the given selector is missing from the page:

```
// Wait a maximum of five seconds until the selector is missing...
$browser->waitUntilMissing('.selector');

// Wait a maximum of one second until the selector is missing...
$browser->waitUntilMissing('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for an element to appear that matches a given selector and then interact with the element. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used to accomplish this. All element operations performed within the given closure will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
          ->press('OK');
});
```

Waiting For Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
```

```
$browser->waitForText('Hello World', 1);
```

You may use the `waitUntilMissingText` method to wait until the displayed text has been removed from the page:

```
// Wait a maximum of five seconds for the text to be removed...
$browser->waitUntilMissingText('Hello World');

// Wait a maximum of one second for the text to be removed...
$browser->waitUntilMissingText('Hello World', 1);
```

Waiting For Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting On The Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to make assertions after a page has been reloaded, use the `waitForReload` method:

```
$browser->click('.some-action')
        ->waitForReload()
        ->assertSee('something');
```

Waiting On JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitUntil` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

Waiting On Vue Expressions

The `waitUntilVue` and `waitUntilVueIsNot` methods may be used to wait until a Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitUntilVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitUntilVueIsNot('user.name', null, '@user');
```

Waiting With A Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given closure to return `true`. The `waitUsing` method accepts the maximum number of

seconds to wait, the interval at which the closure should be evaluated, the closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Scrolling An Element Into View

Sometimes you may not be able to click on an element because it is outside of the viewable area of the browser. The `scrollIntoView` method will scroll the browser window until the element at the given selector is within the view:

```
$browser->scrollIntoView('.selector')
        ->click('.selector');
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

assertTitle assertTitleContains assertUrlIs assertSchemeIs assertSchemeIsNot assertHostIs assertHostIsNot assertPortIs assertPortIsNot assertPathBeginsWith assertPathIs assertPathIsNot assertRouteIs assertQueryStringHas assertQueryStringMissing assertFragmentIs assertFragmentBeginsWith assertFragmentIsNot assertHasCookie assertHasPlainCookie assertCookieMissing assertPlainCookieMissing assertCookieValue assertPlainCookieValue assertSee assertDontSee assertSeeIn assertDontSeeIn assertSeeAnythingIn assertSeeNothingIn assertScript assertSourceHas assertSourceMissing assertSeeLink assertDontSeeLink assertInputValue assertInputValueIsNot assertChecked assertNotChecked assertRadioSelected assertRadioNotSelected assertSelected assertNotSelected assertSelectHasOptions assertSelectMissingOptions assertSelectHasOption assertSelectMissingOption assertValue assertAttribute assertAriaAttribute assertDataAttribute assertVisible assertPresent assertMissing assertDialogOpened assertEnabled assertDisabled assertButtonEnabled assertButtonDisabled assertFocused assertNotFocused assertAuthenticated assertGuest assertAuthenticatedAs assertVue assertVueIsNot assertVueContains assertVueDoesNotContain

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith('/home');
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the URL's current hash fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the URL's current hash fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the URL's current hash fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given encrypted cookie is present:

```
$browser->assertHasCookie($name);
```

assertHasPlainCookie

Assert that the given unencrypted cookie is present:

```
$browser->assertHasPlainCookie($name);
```

assertCookieMissing

Assert that the given encrypted cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertPlainCookieMissing

Assert that the given unencrypted cookie is not present:

```
$browser->assertPlainCookieMissing($name);
```

assertCookieValue

Assert that an encrypted cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSeeAnythingIn

Assert that any text is present within the selector:

```
$browser->assertSeeAnythingIn($selector);
```

assertSeeNothingIn

Assert that no text is present within the selector:

```
$browser->assertSeeNothingIn($selector);
```

assertScript

Assert that the given JavaScript expression evaluates to the given value:

```
$browser->assertScript('window.isLoaded')
        ->assertScript('document.readyState', 'complete');
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValueIsNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertSelectMissingOption

Assert that the given value is not available to be selected:

```
$browser->assertSelectMissingOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertAttribute

Assert that the element matching the given selector has the given value in the provided attribute:

```
$browser->assertAttribute($selector, $attribute, $value);
```

assertAriaAttribute

Assert that the element matching the given selector has the given value in the provided aria attribute:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

For example, given the markup `<button aria-label="Add"></button>`, you may assert against the `aria-label` attribute like so:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

assertDataAttribute

Assert that the element matching the given selector has the given value in the provided data attribute:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

For example, given the markup `<tr id="row-1" data-content="attendees"></tr>`, you may assert against the `data-label` attribute like so:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present:

```
$browser->assertPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertAuthenticated

Assert that the user is authenticated:

```
$browser->assertAuthenticated();
```

assertGuest

Assert that the user is not authenticated:

```
$browser->assertGuest();
```

assertAuthenticatedAs

Assert that the user is authenticated as the given user:

```
$browser->assertAuthenticatedAs($user);
```

assertVue

Dusk even allows you to make assertions on the state of Vue component data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/**
 * A basic Vue test example.
 *
 * @return void
 */
public function testVue()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
                ->assertVue('user.name', 'Taylor', '@profile-component');
    });
}
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

assertVueDoesNotContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Dusk Pages allow you to define expressive actions that may then be performed on a given page via a single method. Pages also allow you to define short-cuts to common selectors for your application or for a single page.

Generating Pages

To generate a page object, execute the `dusk:page` Artisan command. All page objects will be placed in your application's `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be discussed in more detail below.

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**
 * Get the URL for the page.
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. It is not actually necessary to place anything within this method; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

Navigating To Pages

Once a page has been defined, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
        ->clickLink('Create Playlist')
        ->on(new CreatePlaylist)
        ->assertSee('@create');
```

Shorthand Selectors

The `elements` method within page classes allow you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
```

```
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Once the shortcut has been defined, you may use the shorthand selector anywhere you would typically use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  string  $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
                ->check('share')
                ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed as the first argument to custom page methods:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, execute the `dusk:component` Artisan command. New components are placed in the `tests/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  int  $year
     * @param  int  $month
     * @param  int  $day
     * @return void
     */
    public function selectDate(Browser $browser, $year, $month, $day)
    {
        $browser->click('@date-field')
                ->within('@year-list', function ($browser) use ($year) {
                    $browser->click($year);
                })
```

```
                ->within('@month-list', function ($browser) use ($month) {
                    $browser->click($month);
                })
                ->within('@day-list', function ($browser) use ($day) {
                    $browser->click($day);
                });
    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```
<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->within(new DatePicker, function ($browser) {
                        $browser->selectDate(2019, 1, 30);
                    })
                    ->assertSee('January');
        });
    }
}
```

Continuous Integration

> NOTE Most Dusk continuous integration configurations expect your Laravel application to be served using the built-in PHP development server on port 8000. Therefore, before continuing, you should ensure that your continous integration environment has an `APP_URL` environment variable value of `http://127.0.0.1:8000`.

Heroku CI

To run Dusk tests on Heroku CI, add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &' &&
nohup bash -c 'php artisan serve > /dev/null 2>&1 &' && php artisan dusk"
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on Travis CI, use the following `.travis.yml` configuration. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost &
  - php artisan serve &

script:
  - php artisan dusk
```

GitHub Actions

If you are using Github Actions to run your Dusk tests, you may use the following configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
name: CI
on: [push]
jobs:

  dusk-php:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: |
          sudo systemctl start mysql
          mysql --user="root" --password="root" -e "CREATE DATABASE 'my-database' character set UTF8mb4
collate utf8mb4_bin;"
      - name: Install Composer Dependencies
        run: composer install --no-progress --no-suggest --prefer-dist --optimize-autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver `/opt/google/chrome/chrome --version | cut -d " " -f3 | cut
-d "." -f1`
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux &
      - name: Run Laravel Server
        run: php artisan serve &
      - name: Run Dusk Tests
        env:
          APP_URL: "http://127.0.0.1:8000"
        run: php artisan dusk
      - name: Upload Screenshots
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: screenshots
          path: tests/Browser/screenshots
      - name: Upload Console Logs
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: console
          path: tests/Browser/console
```

Packages

Laravel Envoy

- Introduction
- Installation
- Writing Tasks
  - Defining Tasks
  - Multiple Servers
  - Setup
  - Variables
  - Stories
- Running Tasks
  - Confirming Task Execution
- Notifications
  - Slack
  - Discord
  - Telegram

Introduction

Laravel Envoy is a tool for executing common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems. However, Windows support is achievable using WSL2.

Installation

First, install Envoy into your project using the Composer package manager:

```
composer require laravel/envoy --dev
```

Once Envoy has been installed, the Envoy binary will be available in your application's `vendor/bin` directory:

```
php vendor/bin/envoy
```

Writing Tasks

Defining Tasks

Tasks are the basic building block of Envoy. Tasks define the shell commands that should execute on your remote servers when the task is invoked. For example, you might define a task that executes the `php artisan queue:restart` command on all of your application's queue worker servers.

All of your Envoy tasks should be defined in an `Envoy.blade.php` file at the root of your application. Here's an example to get you started:

```
@servers(['web' => ['user@192.168.1.1'], 'workers' => ['user@192.168.1.2']])

@task('restart-queues', ['on' => 'workers'])
    cd /home/user/example.com
    php artisan queue:restart
@endtask
```

As you can see, an array of `@servers` is defined at the top of the file, allowing you to reference these servers via the `on` option of your task declarations. The `@servers` declaration should always be placed on a single line. Within your `@task` declarations, you should place the shell commands that should execute on your servers when the task is invoked.

Local Tasks

You can force a script to run on your local computer by specifying the server's IP address as `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Importing Envoy Tasks

Using the `@import` directive, you may import other Envoy files so their stories and tasks are added to yours. After the files have been imported, you may execute the tasks they contain as if they were defined in your own Envoy file:

```
@import('vendor/package/Envoy.blade.php')
```

Multiple Servers

Envoy allows you to easily run a task across multiple servers. First, add additional servers to your `@servers` declaration. Each server should be assigned a unique name. Once you have defined your additional servers you may list each of the servers in the task's `on` array:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Parallel Execution

By default, tasks will be executed on each server serially. In other words, a task will finish running on the first server before proceeding to execute on the second server. If you would like to run a task across multiple servers in parallel, add the `parallel` option to your task declaration:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Setup

Sometimes, you may need to execute arbitrary PHP code before running your Envoy tasks. You may use the `@setup` directive to define a block of PHP code that should execute before your tasks:

```
@setup
    $now = new DateTime;
@endsetup
```

If you need to require other PHP files before your task is executed, you may use the `@include` directive at the top of your `Envoy.blade.php` file:

```
@include('vendor/autoload.php')

@task('restart-queues')
    # ...
@endtask
```

Variables

If needed, you may pass arguments to Envoy tasks by specifying them on the command line when invoking Envoy:

```
php vendor/bin/envoy run deploy --branch=master
```

You may access the options within your tasks using Blade's "echo" syntax. You may also define Blade `if`

statements and loops within your tasks. For example, let's verify the presence of the `$branch` variable before executing the `git pull` command:

```
@servers(['web' => ['user@192.168.1.1']])

@task('deploy', ['on' => 'web'])
    cd /home/user/example.com

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate --force
@endtask
```

Stories

Stories group a set of tasks under a single, convenient name. For instance, a `deploy` story may run the `update-code` and `install-dependencies` tasks by listing the task names within its definition:

```
@servers(['web' => ['user@192.168.1.1']])

@story('deploy')
    update-code
    install-dependencies
@endstory

@task('update-code')
    cd /home/user/example.com
    git pull origin master
@endtask

@task('install-dependencies')
    cd /home/user/example.com
    composer install
@endtask
```

Once the story has been written, you may invoke it in the same way you would invoke a task:

```
php vendor/bin/envoy run deploy
```

Running Tasks

To run a task or story that is defined in your application's `Envoy.blade.php` file, execute Envoy's `run` command, passing the name of the task or story you would like to execute. Envoy will execute the task and display the output from your remote servers as the task is running:

```
php vendor/bin/envoy run deploy
```

Confirming Task Execution

If you would like to be prompted for confirmation before running a given task on your servers, you should add the `confirm` directive to your task declaration. This option is particularly useful for destructive operations:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Notifications

Slack

Envoy supports sending notifications to Slack after each task is executed. The `@slack` directive accepts a Slack hook URL and a channel / user name. You may retrieve your webhook URL by creating an "Incoming WebHooks" integration in your Slack control panel.

You should pass the entire webhook URL as the first argument given to the `@slack` directive. The second argument given to the `@slack` directive should be a channel name (`#channel`) or a user name (`@user`):

```
@finished
    @slack('webhook-url', '#bots')
@endfinished
```

Discord

Envoy also supports sending notifications to Discord after each task is executed. The `@discord` directive accepts a Discord hook URL and a message. You may retrieve your webhook URL by creating a "Webhook" in your Server Settings and choosing which channel the webhook should post to. You should pass the entire Webhook URL into the `@discord` directive:

```
@finished
    @discord('discord-webhook-url')
@endfinished
```

Telegram

Envoy also supports sending notifications to Telegram after each task is executed. The `@telegram` directive accepts a Telegram Bot ID and a Chat ID. You may retrieve your Bot ID by creating a new bot using BotFather. You can retrieve a valid Chat ID using @username_to_id_bot. You should pass the entire Bot ID and Chat ID into the `@telegram` directive:

```
@finished
    @telegram('bot-id','chat-id')
@endfinished
```

Packages

Laravel Fortify

- Introduction
  - What Is Fortify?
  - When Should I Use Fortify?
- Installation
  - The Fortify Service Provider
  - Fortify Features
  - Disabling Views
- Authentication
  - Customizing User Authentication
- Two Factor Authentication
  - Enabling Two Factor Authentication
  - Authenticating With Two Factor Authentication
  - Disabling Two Factor Authentication
- Registration
  - Customizing Registration
- Password Reset
  - Requesting A Password Reset Link
  - Resetting The Password
  - Customizing Password Resets
- Email Verification
  - Protecting Routes
- Password Confirmation

Introduction

Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more. After installing Fortify, you may run the `route:list` Artisan command to see the routes that Fortify has registered.

Since Fortify does not provide its own user interface, it is meant to be paired with your own user interface which makes requests to the routes it registers. We will discuss exactly how to make requests to these routes in the remainder of this documentation.

> TIP Remember, Fortify is a package that is meant to give you a head start implementing Laravel's authentication features. **You are not required to use it.** You are always free to manually interact with Laravel's authentication services by following the documentation available in the authentication, password reset, and email verification documentation.

What Is Fortify?

As mentioned previously, Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more.

You are not required to use Fortify in order to use Laravel's authentication features. You are always free to manually interact with Laravel's authentication services by following the documentation available in the authentication, password reset, and email verification documentation.

If you are new to Laravel, you may wish to explore the Laravel Breeze application starter kit before attempting to use Laravel Fortify. Laravel Breeze provides an authentication scaffolding for your application that includes a user interface built with Tailwind CSS. Unlike Fortify, Breeze publishes its routes and controllers directly into your application. This allows you to study and get comfortable with Laravel's authentication features before allowing Laravel Fortify to implement these features for you.

Laravel Fortify essentially takes the routes and controllers of Laravel Breeze and offers them as a package that does not include a user interface. This allows you to still quickly scaffold the backend implementation of your application's authentication layer without being tied to any particular frontend opinions.

When Should I Use Fortify?

You may be wondering when it is appropriate to use Laravel Fortify. First, if you are using one of Laravel's application starter kits, you do not need to install Laravel Fortify since all of Laravel's application starter kits already provide a full authentication implementation.

If you are not using an application starter kit and your application needs authentication features, you have two options: manually implement your application's authentication features or use Laravel Fortify to provide the backend implementation of these features.

If you choose to install Fortify, your user interface will make requests to Fortify's authentication routes that are detailed in this documentation in order to authenticate and register users.

If you choose to manually interact with Laravel's authentication services instead of using Fortify, you may do so by following the documentation available in the authentication, password reset, and email verification documentation.

Laravel Fortify & Laravel Sanctum

Some developers become confused regarding the difference between Laravel Sanctum and Laravel Fortify. Because the two packages solve two different but related problems, Laravel Fortify and Laravel Sanctum are not mutually exclusive or competing packages.

Laravel Sanctum is only concerned with managing API tokens and authenticating existing users using session cookies or tokens. Sanctum does not provide any routes that handle user registration, password reset, etc.

If you are attempting to manually build the authentication layer for an application that offers an API or serves as the backend for a single-page application, it is entirely possible that you will utilize both Laravel Fortify (for user registration, password reset, etc.) and Laravel Sanctum (API token management, session authentication).

Installation

To get started, install Fortify using the Composer package manager:

```
composer require laravel/fortify
```

Next, publish Fortify's resources using the `vendor:publish` command:

```
php artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProvider"
```

This command will publish Fortify's actions to your `app/Actions` directory, which will be created if it does not exist. In addition, Fortify's configuration file and migrations will be published.

Next, you should migrate your database:

```
php artisan migrate
```

The Fortify Service Provider

The `vendor:publish` command discussed above will also publish the `App\Providers\FortifyServiceProvider` class. You should ensure this class is registered within the `providers` array of your application's `config/app.php` configuration file.

The Fortify service provider registers the actions that Fortify published and instructs Fortify to use them when their respective tasks are executed by Fortify.

Fortify Features

The `fortify` configuration file contains a `features` configuration array. This array defines which backend routes / features Fortify will expose by default. If you are not using Fortify in combination with Laravel Jetstream, we recommend that you only enable the following features, which are the basic authentication features provided by most Laravel applications:

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
],
```

Disabling Views

By default, Fortify define routes that are intended to return views, such as a login screen or registration screen. However, if you are building a JavaScript driven single-page application, you may not need these routes. For that reason, you may disable these routes entirely by setting the `views` configuration value within your application's `config/fortify.php` configuration file to `false`:

```
'views' => false,
```

Disabling Views & Password Reset

If you choose to disable Fortify's views and you will be implementing password reset features for your application, you should still define a route named `password.reset` that is responsible for displaying your application's "reset password" view. This is necessary because Laravel's `Illuminate\Auth\Notifications\ResetPassword` notification will generate the password reset URL via the `password.reset` named route.

Authentication

To get started, we need to instruct Fortify how to return our "login" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that is already completed for you, you should use an application starter kit.

All of the authentication view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class. Fortify will take care of defining the `/login` route that returns this view:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::loginView(fn () => view('auth.login'));

    // ...
}
```

Your login template should include a form that makes a POST request to `/login`. The `/login` endpoint expects a string email address / username and a `password`. The name of the email / username field should match the `username` value within the `config/fortify.php` configuration file. In addition, a boolean `remember` field may be provided to indicate that the user would like to use the "remember me" functionality provided by Laravel.

If the login attempt is successful, Fortify will redirect you to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the login screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with the 422 HTTP response.

Customizing User Authentication

Fortify will automatically retrieve and authenticate the user based on the provided credentials and the authentication guard that is configured for your application. However, you may sometimes wish to have full customization over how login credentials are authenticated and users are retrieved. Thankfully, Fortify allows you to easily accomplish this using the `Fortify::authenticateUsing` method.

This method accepts a closure which receives the incoming HTTP request. The closure is responsible for validating the login credentials attached to the request and returning the associated user instance. If the credentials are invalid or no user can be found, `null` or `false` should be returned by the closure. Typically, this method should be called from the `boot` method of your `FortifyServiceProvider`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::authenticateUsing(function (Request $request) {
        $user = User::where('email', $request->email)->first();

        if ($user &&
            Hash::check($request->password, $user->password)) {
            return $user;
        }
    });

    // ...
}
```

Authentication Guard

You may customize the authentication guard used by Fortify within your application's `fortify` configuration file. However, you should ensure that the configured guard is an implementation of `Illuminate\Contracts\Auth\StatefulGuard`. If you are attempting to use Laravel Fortify to authenticate an SPA, you should use Laravel's default `web` guard in combination with Laravel Sanctum.

Two Factor Authentication

When Fortify's two factor authentication feature is enabled, the user is required to input a six digit numeric token during the authentication process. This token is generated using a time-based one-time password (TOTP) that can be retrieved from any TOTP compatible mobile authentication application such as Google Authenticator.

Before getting started, you should first ensure that your application's `App\Models\User` model uses the `Laravel\Fortify\TwoFactorAuthenticatable` trait:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Fortify\TwoFactorAuthenticatable;

class User extends Authenticatable
{
    use Notifiable, TwoFactorAuthenticatable;
}
```

Next, you should build a screen within your application where users can manage their two factor authentication settings. This screen should allow the user to enable and disable two factor authentication, as well as regenerate their two factor authentication recovery codes.

By default, the `features` array of the `fortify` configuration file instructs Fortify's two factor authentication settings to require password confirmation before modification. Therefore, your application should implement Fortify's password confirmation feature before continuing.

Enabling Two Factor Authentication

To enable two factor authentication, your application should make a POST request to the `/user/two-factor-authentication` endpoint defined by Fortify. If the request is successful, the user will be redirected back to the previous URL and the `status` session variable will be set to `two-factor-authentication-enabled`. You may detect this `status` session variable within your templates to display the appropriate success message. If the request was an XHR request, `200` HTTP response will be returned:

```
@if (session('status') == 'two-factor-authentication-enabled')
    <div class="mb-4 font-medium text-sm text-green-600">
        Two factor authentication has been enabled.
    </div>
@endif
```

Next, you should display the two factor authentication QR code for the user to scan into their authenticator application. If you are using Blade to render your application's frontend, you may retrieve the QR code SVG using the `twoFactorQrCodeSvg` method available on the user instance:

```
$request->user()->twoFactorQrCodeSvg();
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-qr-code` endpoint to retrieve the user's two factor authentication QR code. This endpoint will return a JSON object containing an `svg` key.

Displaying The Recovery Codes

You should also display the user's two factor recovery codes. These recovery codes allow the user to authenticate if they lose access to their mobile device. If you are using Blade to render your application's frontend, you may access the recovery codes via the authenticated user instance:

```
(array) $request->user()->two_factor_recovery_codes
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-recovery-codes` endpoint. This endpoint will return a JSON array containing the user's recovery codes.

To regenerate the user's recovery codes, your application should make a POST request to the `/user/two-factor-recovery-codes` endpoint.

Authenticating With Two Factor Authentication

During the authentication process, Fortify will automatically redirect the user to your application's two factor authentication challenge screen. However, if your application is making an XHR login request, the JSON response returned after a successful authentication attempt will contain a JSON object that has a `two_factor` boolean property. You should inspect this value to know whether you should redirect to your application's two factor authentication challenge screen.

To begin implementing two factor authentication functionality, we need to instruct Fortify how to return our two factor authentication challenge view. All of Fortify's authentication view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

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

```
        return view('auth.two-factor-challenge');
    });

    // ...
}
```

Fortify will take care of defining the `/two-factor-challenge` route that returns this view. Your `two-factor-challenge` template should include a form that makes a POST request to the `/two-factor-challenge` endpoint. The `/two-factor-challenge` action expects a `code` field that contains a valid TOTP token or a `recovery_code` field that contains one of the user's recovery codes.

If the login attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 204 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the login screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Disabling Two Factor Authentication

To disable two factor authentication, your application should make a DELETE request to the `/user/two-factor-authentication` endpoint. Remember, Fortify's two factor authentication endpoints require password confirmation prior to being called.

Registration

To begin implementing our application's registration functionality, we need to instruct Fortify how to return our "register" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that is already completed for you, you should use an application starter kit.

All of the Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::registerView(fn () => view('auth.register'));

    // ...
}
```

Fortify will take care of defining the `/register` route that returns this view. Your `register` template should include a form that makes a POST request to the `/register` endpoint defined by Fortify.

The `/register` endpoint expects a string `name`, string email address / username, `password`, and `password_confirmation` fields. The name of the email / username field should match the `username` configuration value defined within your application's `fortify` configuration file.

If the registration attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the registration screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Registration

The user validation and creation process may be customized by modifying the `App\Actions\Fortify\CreateNewUser` action that was generated when you installed Laravel Fortify.

Password Reset

Requesting A Password Reset Link

To begin implementing our application's password reset functionality, we need to instruct Fortify how to return our "forgot password" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that is already completed for you, you should use an application starter kit.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::requestPasswordResetLinkView(function () {
        return view('auth.forgot-password');
    });

    // ...
}
```

Fortify will take care of defining the `/forgot-password` endpoint that returns this view. Your `forgot-password` template should include a form that makes a POST request to the `/forgot-password` endpoint.

The `/forgot-password` endpoint expects a string `email` field. The name of this field / database column should match the `email` configuration value within your application's `fortify` configuration file.

Handling The Password Reset Link Request Response

If the password reset link request was successful, Fortify will redirect the user back to the `/forgot-password` endpoint and send an email to the user with a secure link they can use to reset their password. If the request was an XHR request, a 200 HTTP response will be returned.

After being redirected back to the `/forgot-password` endpoint after a successful request, the `status` session variable may be used to display the status of the password reset link request attempt. The value of this session variable will match one of the translation strings defined within your application's `passwords` language file:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was not successful, the user will be redirected back to the request password reset link screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Resetting The Password

To finish implementing our application's password reset functionality, we need to instruct Fortify how to return our "reset password" view.

All of Fortify's view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::resetPasswordView(function ($request) {
        return view('auth.reset-password', ['request' => $request]);
    });

    // ...
}
```

Fortify will take care of defining the route to display this view. Your `reset-password` template should include a form that makes a POST request to `/reset-password`.

The `/reset-password` endpoint expects a string `email` field, a `password` field, a `password_confirmation` field, and a hidden field named `token` that contains the value of `request()->route('token')`. The name of the "email" field / database column should match the `email` configuration value defined within your application's `fortify` configuration file.

Handling The Password Reset Response

If the password reset request was successful, Fortify will redirect back to the `/login` route so that the user can login with their new password. In addition a `status` session variable will be set so that you may display the successful status of the reset on your login screen:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the reset password screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Password Resets

The password reset process may be customized by modifying the `App\Actions\ResetUserPassword` action that was generated when you installed Laravel Fortify.

Email Verification

After registration, you may wish for users to verify their email address before they continue accessing your application. To get started, ensure the `emailVerification` feature is enabled in your `fortify` configuration file's `features` array. Next, you should ensure that your `App\Models\User` class implements the `Illuminate\Contracts\Auth\MustVerifyEmail` interface.

Once these two setup steps have been completed, newly registered users will receive an email prompting them to verify their email address ownership. However, we need to inform Fortify how to display the email verification screen which informs the user that they need to go click the verification link in the email.

All of Fortify's view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::verifyEmailView(fn () => view('auth.verify-email'));

    // ...
}
```

Fortify will take care of defining the route that displays this view when a user is redirected to the `/email/verify` endpoint by Laravel's built-in `verified` middleware.

Your `verify-email` template should include an informational message instructing the user to click the email verification link that was sent to their email address.

Resending Email Verification Links

If you wish, you may add a button to your application's `verify-email` template that triggers a POST request to the `/email/verification-notification` endpoint. When this endpoint receives a request, a new verification email link will be emailed to the user, allowing the user to get a new verification link if the previous one was accidentally deleted or lost.

If the request to resend the verification link email was successful, Fortify will redirect the user back to the `/email/verify` endpoint with a `status` session variable, allowing you to display an informational message to the user informing them the operation was successful. If the request was an XHR request, a 202 HTTP response will be returned:

```
@if (session('status') == 'verification-link-sent')
    <div class="mb-4 font-medium text-sm text-green-600">
        A new email verification link has been emailed to you!
    </div>
@endif
```

Protecting Routes

To specify that a route or group of routes requires that the user has verified their email address, you should attach Laravel's built-in `verified` middleware to the route. This middleware is registered within your application's `App\Http\Kernel` class:

```
Route::get('/dashboard', function () {
    // ...
})->middleware(['verified']);
```

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed. Typically, these routes are protected by Laravel's built-in `password.confirm` middleware.

To begin implementing password confirmation functionality, we need to instruct Fortify how to return our application's "password confirmation" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that is already completed for you, you should use an application starter kit.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
```

```
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::confirmPasswordView(function () {
        return view('auth.confirm-password');
    });

    // ...
}
```

Fortify will take care of defining the `/user/confirm-password` endpoint that returns this view. Your `confirm-password` template should include a form that makes a POST request to the `/user/confirm-password` endpoint. The `/user/confirm-password` endpoint expects a `password` field that contains the user's current password.

If the password matches the user's current password, Fortify will redirect the user to the route they were attempting to access. If the request was an XHR request, a 201 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the confirm password screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Packages

Laravel Homestead

- Introduction
- Installation & Setup
  - First Steps
  - Configuring Homestead
  - Configuring Nginx Sites
  - Configuring Services
  - Launching The Vagrant Box
  - Per Project Installation
  - Installing Optional Features
  - Aliases
- Updating Homestead
- Daily Usage
  - Connecting Via SSH
  - Adding Additional Sites
  - Environment Variables
  - Ports
  - PHP Versions
  - Connecting To Databases
  - Database Backups
  - Database Snapshots
  - Configuring Cron Schedules
  - Configuring MailHog
  - Configuring Minio
  - Laravel Dusk
  - Sharing Your Environment
- Debugging & Profiling
  - Debugging Web Requests With Xdebug
  - Debugging CLI Applications
  - Profiling Applications with Blackfire
- Network Interfaces
- Extending Homestead
- Provider Specific Settings
  - VirtualBox

Introduction

Laravel strives to make the entire PHP development experience delightful, including your local development environment. Laravel Homestead is an official, pre-packaged Vagrant box that provides you a wonderful development environment without requiring you to install PHP, a web server, and any other server software on your local machine.

Vagrant provides a simple, elegant way to manage and provision Virtual Machines. Vagrant boxes are completely disposable. If something goes wrong, you can destroy and re-create the box in minutes!

Homestead runs on any Windows, macOS, or Linux system and includes Nginx, PHP, MySQL, PostgreSQL, Redis, Memcached, Node, and all of the other software you need to develop amazing Laravel applications.

> NOTE If you are using Windows, you may need to enable hardware virtualization (VT-x). It can usually be enabled via your BIOS. If you are using Hyper-V on a UEFI system you may additionally need to disable Hyper-V in order to access VT-x.

Included Software

- Ubuntu 18.04 (`master` branch)
- Ubuntu 20.04 (`20.04` branch)

- Git
- PHP 8.0
- PHP 7.4
- PHP 7.3
- PHP 7.2
- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- MySQL
- lmm
- Sqlite3
- PostgreSQL (9.6, 10, 11, 12)
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- avahi
- ngrok
- Xdebug
- XHProf / Tideways / XHGui
- wp-cli

Optional Software

- Apache
- Blackfire
- Cassandra
- Chronograf
- CouchDB
- Crystal & Lucky Framework
- Docker
- Elasticsearch
- Gearman
- Go
- Grafana
- InfluxDB
- MariaDB
- MinIO
- MongoDB
- MySQL 8
- Neo4j
- Oh My Zsh
- Open Resty
- PM2
- Python
- RabbitMQ
- Solr
- Webdriver & Laravel Dusk Utilities

Installation & Setup

First Steps

Before launching your Homestead environment, you must install Vagrant as well as one of the following supported providers:

- VirtualBox 6.1.x
- VMWare
- Parallels
- Hyper-V

All of these software packages provide easy-to-use visual installers for all popular operating systems.

To use the VMware provider, you will need to purchase both VMware Fusion / Workstation and the VMware Vagrant plug-in. Though it is not free, VMware can provide faster shared folder performance out of the box.

To use the Parallels provider, you will need to install Parallels Vagrant plug-in. It is free of charge.

Because of Vagrant limitations, the Hyper-V provider ignores all networking settings.

Installing The Homestead Vagrant Box

Once VirtualBox / VMware and Vagrant have been installed, you should add the `laravel/homestead` box to your Vagrant installation by executing the following command in your terminal. It will take a few minutes to download the box, depending on your Internet connection speed:

```
vagrant box add laravel/homestead
```

If this command fails, make sure your Vagrant installation is up to date.

> NOTE Homestead periodically issues "alpha" / "beta" boxes for testing, which may interfere with the `vagrant box add` command. If you are having issues running `vagrant box add`, you may run the `vagrant up` command and the correct box will be downloaded when Vagrant attempts to start the virtual machine.

Installing Homestead

You may install Homestead by cloning the Homstead repository onto your host machine. Consider cloning the repository into a `Homestead` folder within your "home" directory, as the Homestead virtual machine will serve as the host to all of your Laravel applications. Throughout this documentation, we will refer to this directory as your "Homestead directory":

```
git clone https://github.com/laravel/homestead.git ~/Homestead
```

After cloning the Laravel Homestead repository, you should checkout the `release` branch. This branch always contains the latest stable release of Homestead:

```
cd ~/Homestead

git checkout release
```

Next, execute the `bash init.sh` command from the Homestead directory to create the `Homestead.yaml` configuration file. The `Homestead.yaml` file is where you will configure all of the settings for your Homestead installation. This file will be placed in the Homestead directory:

```
// macOS / Linux...
bash init.sh

// Windows...
init.bat
```

Configuring Homestead

Setting Your Provider

The `provider` key in your `Homestead.yaml` file indicates which Vagrant provider should be used: `virtualbox`, `vmware_fusion`, `vmware_workstation`, `parallels`, or `hyperv`:

```
provider: virtualbox
```

Configuring Shared Folders

The `folders` property of the `Homestead.yaml` file lists all of the folders you wish to share with your Homestead environment. As files within these folders are changed, they will be kept in sync between your local machine and the Homestead virtual environment. You may configure as many shared folders as necessary:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
```

> NOTE Windows users should not use the `~/` path syntax and instead should use the full path to their project, such as `C:\Users\user\Code\project1`.

You should always map individual applications to their own folder mapping instead of mapping a single large directory that contains all of your applications. When you map a folder, the virtual machine must keep track of all disk IO for *every* file in the folder. You may experience reduced performance if you have a large number of files in a folder:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
    - map: ~/code/project2
      to: /home/vagrant/project2
```

> NOTE You should never mount `.` (the current directory) when using Homestead. This causes Vagrant to not map the current folder to `/vagrant` and will break optional features and cause unexpected results while provisioning.

To enable NFS, you may add a `type` option to your folder mapping:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
      type: "nfs"
```

> NOTE When using NFS on Windows, you should consider installing the vagrant-winnfsd plug-in. This plug-in will maintain the correct user / group permissions for files and directories within the Homestead virtual machine.

You may also pass any options supported by Vagrant's Synced Folders by listing them under the `options` key:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
      type: "rsync"
      options:
          rsync__args: ["--verbose", "--archive", "--delete", "-zz"]
          rsync__exclude: ["node_modules"]
```

Configuring Nginx Sites

Not familiar with Nginx? No problem. Your `Homstead.yaml` file's `sites` property allows you to easily map a "domain" to a folder on your Homestead environment. A sample site configuration is included in the `Homestead.yaml` file. Again, you may add as many sites to your Homestead environment as necessary. Homestead can serve as a convenient, virtualized environment for every Laravel application you are working on:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
```

If you change the `sites` property after provisioning the Homestead virtual machine, you should execute the `vagrant reload --provision` command in your terminal to update the Nginx configuration on the virtual machine.

> NOTE Homestead scripts are built to be as idempotent as possible. However, if you are experiencing issues while provisioning you should destroy and rebuild the machine by executing the `vagrant destroy && vagrant up` command.

Hostname Resolution

Homestead publishes hostnames using `mDNS` for automatic host resolution. If you set `hostname: homestead` in your `Homestead.yaml` file, the host will be available at `homestead.local`. macOS, iOS, and Linux desktop distributions include `mDNS` support by default. If you are using Windows, you must install Bonjour Print Services for Windows.

Using automatic hostnames works best for per project installations of Homestead. If you host multiple sites on a single Homestead instance, you may add the "domains" for your web sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead sites into your Homestead virtual machine. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

```
192.168.10.10  homestead.test
```

Make sure the IP address listed is the one set in your `Homestead.yaml` file. Once you have added the domain to your `hosts` file and launched the Vagrant box you will be able to access the site via your web browser:

```
http://homestead.test
```

Configuring Services

Homestead starts several services by default; however, you may customize which services are enabled or disabled during provisioning. For example, you may enable PostgreSQL and disable MySQL by modifying the `services` option within your `Homestead.yaml` file:

```
services:
    - enabled:
        - "postgresql@12-main"
    - disabled:
        - "mysql"
```

The specified services will be started or stopped based on their order in the `enabled` and `disabled` directives.

Launching The Vagrant Box

Once you have edited the `Homestead.yaml` to your liking, run the `vagrant up` command from your Homestead directory. Vagrant will boot the virtual machine and automatically configure your shared folders and Nginx sites.

To destroy the machine, you may use the `vagrant destroy --force` command.

Per Project Installation

Instead of installing Homestead globally and sharing the same Homestead virtual machine across all of your projects, you may instead configure a Homestead instance for each project you manage. Installing Homestead per project may be beneficial if you wish to ship a `Vagrantfile` with your project, allowing others working on the project to `vagrant up` immediately after cloning the project's repository.

You may install Homestead into your project using the Composer package manger:

```
composer require laravel/homestead --dev
```

Once Homestead has been installed, invoke Homestead's `make` command to generate the `Vagrantfile` and `Homestead.yaml` file for your project. These files will be placed in the root of your project. The `make` command will automatically configure the `sites` and `folders` directives in the `Homestead.yaml` file:

```
// macOS / Linux...
php vendor/bin/homestead make

// Windows...
vendor\\bin\\homestead make
```

Next, run the `vagrant up` command in your terminal and access your project at `http://homestead.test` in your

browser. Remember, you will still need to add an `/etc/hosts` file entry for `homestead.test` or the domain of your choice if you are not using automatic hostname resolution.

Installing Optional Features

Optional software is installed using the `features` option within your `Homestead.yaml` file. Most features can be enabled or disabled with a boolean value, while some features allow multiple configuration options:

```
features:
    - blackfire:
        server_id: "server_id"
        server_token: "server_value"
        client_id: "client_id"
        client_token: "client_value"
    - cassandra: true
    - chronograf: true
    - couchdb: true
    - crystal: true
    - docker: true
    - elasticsearch:
        version: 7.9.0
    - gearman: true
    - golang: true
    - grafana: true
    - influxdb: true
    - mariadb: true
    - minio: true
    - mongodb: true
    - mysql8: true
    - neo4j: true
    - ohmyzsh: true
    - openresty: true
    - pm2: true
    - python: true
    - rabbitmq: true
    - solr: true
    - webdriver: true
```

Elasticsearch

You may specify a supported version of Elasticsearch, which must be an exact version number (major.minor.patch). The default installation will create a cluster named 'homestead'. You should never give Elasticsearch more than half of the operating system's memory, so make sure your Homestead virtual machine has at least twice the Elasticsearch allocation.

TIP Check out the Elasticsearch documentation to learn how to customize your configuration.

MariaDB

Enabling MariaDB will remove MySQL and install MariaDB. MariaDB typically serves as a drop-in replacement for MySQL, so you should still use the `mysql` database driver in your application's database configuration.

MongoDB

The default MongoDB installation will set the database username to `homestead` and the corresponding password to `secret`.

Neo4j

The default Neo4j installation will set the database username to `homestead` and the corresponding password to `secret`. To access the Neo4j browser, visit `http://homestead.test:7474` via your web browser. The ports `7687` (Bolt), `7474` (HTTP), and `7473` (HTTPS) are ready to serve requests from the Neo4j client.

Aliases

You may add Bash aliases to your Homestead virtual machine by modifying the `aliases` file within your Homestead directory:

```
alias c='clear'
alias ..='cd ..'
```

After you have updated the `aliases` file, you should re-provision the Homestead virtual machine using the `vagrant reload --provision` command. This will ensure that your new aliases are available on the machine.

Updating Homestead

Before you begin updating Homestead you should ensure you have removed your current virtual machine by running the following command in your Homestead directory:

```
vagrant destroy
```

Next, you need to update the Homestead source code. If you cloned the repository, you can execute the following commands at the location you originally cloned the repository:

```
git fetch
```

```
git pull origin release
```

These commands pull the latest Homestead code from the GitHub repository, fetch the latest tags, and then check out the latest tagged release. You can find the latest stable release version on Homestead's GitHub releases page.

If you have installed Homestead via your project's `composer.json` file, you should ensure your `composer.json` file contains `"laravel/homestead": "^11"` and update your dependencies:

```
composer update
```

Next, you should update the Vagrant box using the `vagrant box update` command:

```
vagrant box update
```

After updating the Vagrant box, you should run the `bash init.sh` command from the Homestead directory in order to update Homestead's additional configuration files. You will be asked whether you wish to overwrite your existing `Homestead.yaml`, `after.sh`, and `aliases` files:

```
// macOS / Linux...
bash init.sh

// Windows...
init.bat
```

Finally, you will need to regenerate your Homestead virtual machine to utilize the latest Vagrant installation:

```
vagrant up
```

Daily Usage

Connecting Via SSH

You can SSH into your virtual machine by executing the `vagrant ssh` terminal command from your Homestead directory.

Adding Additional Sites

Once your Homestead environment is provisioned and running, you may want to add additional Nginx sites for your other Laravel projects. You can run as many Laravel projects as you wish on a single Homestead environment. To add an additional site, add the site to your `Homestead.yaml` file.

```
sites:
    - map: homestead.test
```

```
      to: /home/vagrant/project1/public
    - map: another.test
      to: /home/vagrant/project2/public
```

> NOTE You should ensure that you have configured a folder mapping for the project's directory before adding the site.

If Vagrant is not automatically managing your "hosts" file, you may need to add the new site to that file as well. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`:

```
192.168.10.10  homestead.test
192.168.10.10  another.test
```

Once the site has been added, execute the `vagrant reload --provision` terminal command from your Homestead directory.

Site Types

Homestead supports several "types" of sites which allow you to easily run projects that are not based on Laravel. For example, we may easily add a Statamic application to Homestead using the `statamic` site type:

```
sites:
    - map: statamic.test
      to: /home/vagrant/my-symfony-project/web
      type: "statamic"
```

The available site types are: `apache`, `apigility`, `expressive`, `laravel` (the default), `proxy`, `silverstripe`, `statamic`, `symfony2`, `symfony4`, and `zf`.

Site Parameters

You may add additional Nginx `fastcgi_param` values to your site via the `params` site directive:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      params:
          - key: FOO
            value: BAR
```

Environment Variables

You can define global environment variables by adding them to your `Homestead.yaml` file:

```
variables:
    - key: APP_ENV
      value: local
    - key: FOO
      value: bar
```

After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing the `vagrant reload --provision` command. This will update the PHP-FPM configuration for all of the installed PHP versions and also update the environment for the `vagrant` user.

Ports

By default, the following ports are forwarded to your Homestead environment:

- **SSH:** 2222 → Forwards To 22
- **ngrok UI:** 4040 → Forwards To 4040
- **HTTP:** 8000 → Forwards To 80
- **HTTPS:** 44300 → Forwards To 443
- **MySQL:** 33060 → Forwards To 3306
- **PostgreSQL:** 54320 → Forwards To 5432
- **MongoDB:** 27017 → Forwards To 27017

- **Mailhog:** 8025 → Forwards To 8025
- **Minio:** 9600 → Forwards To 9600

Forwarding Additional Ports

If you wish, you may forward additional ports to the Vagrant box by defining a `ports` configuration entry within your `Homestead.yaml` file. After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing the `vagrant reload --provision` command:

```
ports:
    - send: 50000
      to: 5000
    - send: 7777
      to: 777
      protocol: udp
```

PHP Versions

Homestead 6 introduced support for running multiple versions of PHP on the same virtual machine. You may specify which version of PHP to use for a given site within your `Homestead.yaml` file. The available PHP versions are: "5.6", "7.0", "7.1", "7.2", "7.3", and "7.4" (the default):

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      php: "7.1"
```

Within your Homestead virtual machine, you may use any of the supported PHP versions via the CLI:

```
php5.6 artisan list
php7.0 artisan list
php7.1 artisan list
php7.2 artisan list
php7.3 artisan list
php7.4 artisan list
```

You may change the default version of PHP used by the CLI by issuing the following commands from within your Homestead virtual machine:

```
php56
php70
php71
php72
php73
php74
```

Connecting To Databases

A `homestead` database is configured for both MySQL and PostgreSQL out of the box. To connect to your MySQL or PostgreSQL database from your host machine's database client, you should connect to `127.0.0.1` on port `33060` (MySQL) or `54320` (PostgreSQL). The username and password for both databases is `homestead` / `secret`.

> NOTE You should only use these non-standard ports when connecting to the databases from your host machine. You will use the default 3306 and 5432 ports in your Laravel application's `database` configuration file since Laravel is running *within* the virtual machine.

Database Backups

Homestead can automatically backup your database when your Homestead virtual machine is destroyed. To utilize this feature, you must be using Vagrant 2.1.0 or greater. Or, if you are using an older version of Vagrant, you must install the `vagrant-triggers` plug-in. To enable automatic database backups, add the following line to your `Homestead.yaml` file:

```
backup: true
```

Once configured, Homestead will export your databases to `mysql_backup` and `postgres_backup` directories when the `vagrant destroy` command is executed. These directories can be found in the folder where you installed Homestead or in the root of your project if you are using the per project installation method.

Database Snapshots

Homestead supports freezing the state of MySQL and MariaDB databases and branching between them using Logical MySQL Manager. For example, imagine working on a site with a multi-gigabyte database. You can import the database and take a snapshot. After doing some work and creating some test content locally, you may quickly restore back to the original state.

Under the hood, LMM uses LVM's thin snapshot functionality with copy-on-write support. In practice, this means that changing a single row in a table will only cause the changes you made to be written to disk, saving significant time and disk space during restores.

Since LMM interacts with LVM, it must be run as `root`. To see all available commands, run the `sudo lmm` command within Vagrant box. A common workflow looks like the following:

- Import a database into the default `master` lmm branch.
- Save a snapshot of the unchanged database using `sudo lmm branch prod-YYYY-MM-DD`.
- Modify the database.
- Run `sudo lmm merge prod-YYYY-MM-DD` to undo all changes.
- Run `sudo lmm delete <branch>` to delete unneeded branches.

Configuring Cron Schedules

Laravel provides a convenient way to schedule cron jobs by scheduling a single `schedule:run` Artisan command to run every minute. The `schedule:run` command will examine the job schedule defined in your `App\Console\Kernel` class to determine which scheduled tasks to run.

If you would like the `schedule:run` command to be run for a Homestead site, you may set the `schedule` option to `true` when defining the site:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      schedule: true
```

The cron job for the site will be defined in the `/etc/cron.d` directory of the Homestead virtual machine.

Configuring MailHog

MailHog allows you to intercept your outgoing email and examine it without actually sending the mail to its recipients. To get started, update your application's `.env` file to use the following mail settings:

```
MAIL_MAILER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Once MailHog has been configured, you may access the MailHog dashboard at `http://localhost:8025`.

Configuring Minio

Minio is an open source object storage server with an Amazon S3 compatible API. To install Minio, update your `Homestead.yaml` file with the following configuration option in the features section:

```
minio: true
```

By default, Minio is available on port 9600. You may access the Minio control panel by visiting `http://localhost:9600`. The default access key is `homestead`, while the default secret key is `secretkey`. When

accessing Minio, you should always use region `us-east-1`.

In order to use Minio you will need to adjust the S3 disk configuration in your application's `config/filesystems.php` configuration file. You will need to add the `use_path_style_endpoint` option to the disk configuration as well as change the `url` key to `endpoint`:

```
's3' => [
    'driver' => 's3',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION'),
    'bucket' => env('AWS_BUCKET'),
    'endpoint' => env('AWS_URL'),
    'use_path_style_endpoint' => true,
]
```

Finally, ensure your `.env` file has the following options:

```
AWS_ACCESS_KEY_ID=homestead
AWS_SECRET_ACCESS_KEY=secretkey
AWS_DEFAULT_REGION=us-east-1
AWS_URL=http://localhost:9600
```

To provision Minio powered "S3" buckets, add a `buckets` directive to your `Homestead.yaml` file. After defining your buckets, you should execute the `vagrant reload --provision` command in your terminal:

```
buckets:
    - name: your-bucket
      policy: public
    - name: your-private-bucket
      policy: none
```

Supported `policy` values include: `none`, `download`, `upload`, and `public`.

Laravel Dusk

In order to run Laravel Dusk tests within Homestead, you should enable the `webdriver` feature in your Homestead configuration:

```
features:
    - webdriver: true
```

After enabling the `webdriver` feature, you should execute the `vagrant reload --provision` command in your terminal.

Sharing Your Environment

Sometimes you may wish to share what you're currently working on with coworkers or a client. Vagrant has built-in support for this via the `vagrant share` command; however, this will not work if you have multiple sites configured in your `Homestead.yaml` file.

To solve this problem, Homestead includes its own `share` command. To get started, SSH into your Homestead virtual machine via `vagrant ssh` and execute the `share homestead.test` command. This command will share the `homestead.test` site from your `Homestead.yaml` configuration file. You may substitute any of your other configured sites for `homestead.test`:

```
share homestead.test
```

After running the command, you will see an Ngrok screen appear which contains the activity log and the publicly accessible URLs for the shared site. If you would like to specify a custom region, subdomain, or other Ngrok runtime option, you may add them to your `share` command:

```
share homestead.test -region=eu -subdomain=laravel
```

> NOTE Remember, Vagrant is inherently insecure and you are exposing your virtual machine to the Internet when running the `share` command.

Debugging & Profiling

Debugging Web Requests With Xdebug

Homestead includes support for step debugging using Xdebug. For example, you can access a page in your browser and PHP will connect to your IDE to allow inspection and modification of the running code.

By default, Xdebug is already running and ready to accept connections. If you need to enable Xdebug on the CLI, execute the `sudo phpenmod xdebug` command within your Homestead virtual machine. Next, follow your IDE's instructions to enable debugging. Finally, configure your browser to trigger Xdebug with an extension or bookmarklet.

> NOTE Xdebug causes PHP to run significantly slower. To disable Xdebug, run `sudo phpdismod xdebug` within your Homestead virtual machine and restart the FPM service.

Autostarting Xdebug

When debugging functional tests that make requests to the web server, it is easier to autostart debugging rather than modifying tests to pass through a custom header or cookie to trigger debugging. To force Xdebug to start automatically, modify the `/etc/php/7.x/fpm/conf.d/20-xdebug.ini` file inside your Homestead virtual machine and add the following configuration:

```
; If Homestead.yaml contains a different subnet for the IP address, this address may be different...
xdebug.remote_host = 192.168.10.1
xdebug.remote_autostart = 1
```

Debugging CLI Applications

To debug a PHP CLI application, use the `xphp` shell alias inside your Homestead virtual machine:

```
xphp /path/to/script
```

Profiling Applications with Blackfire

Blackfire is a service for profiling web requests and CLI applications. It offers an interactive user interface which displays profile data in call-graphs and timelines. It is built for use in development, staging, and production, with no overhead for end users. In addition, Blackfire provides performance, quality, and security checks on code and `php.ini` configuration settings.

The Blackfire Player is an open-source Web Crawling, Web Testing, and Web Scraping application which can work jointly with Blackfire in order to script profiling scenarios.

To enable Blackfire, use the "features" setting in your Homestead configuration file:

```
features:
    - blackfire:
        server_id: "server_id"
        server_token: "server_value"
        client_id: "client_id"
        client_token: "client_value"
```

Blackfire server credentials and client credentials require a Blackfire account. Blackfire offers various options to profile an application, including a CLI tool and browser extension. Please review the Blackfire documentation for more details.

Network Interfaces

The `networks` property of the `Homestead.yaml` file configures network interfaces for your Homestead virtual machine. You may configure as many interfaces as necessary:

```
networks:
    - type: "private_network"
```

```
      ip: "192.168.10.20"
```

To enable a bridged interface, configure a `bridge` setting for the network and change the network type to `public_network`:

```
networks:
    - type: "public_network"
      ip: "192.168.10.20"
      bridge: "en1: Wi-Fi (AirPort)"
```

To enable DHCP, just remove the `ip` option from your configuration:

```
networks:
    - type: "public_network"
      bridge: "en1: Wi-Fi (AirPort)"
```

Extending Homestead

You may extend Homestead using the `after.sh` script in the root of your Homestead directory. Within this file, you may add any shell commands that are necessary to properly configure and customize your virtual machine.

When customizing Homestead, Ubuntu may ask you if you would like to keep a package's original configuration or overwrite it with a new configuration file. To avoid this, you should use the following command when installing packages in order to avoid overwriting any configuration previously written by Homestead:

```
sudo apt-get -y \
    -o Dpkg::Options::="--force-confdef" \
    -o Dpkg::Options::="--force-confold" \
    install package-name
```

User Customizations

When using Homestead with your team, you may want to tweak Homestead to better fit your personal development style. To accomplish this, you may create a `user-customizations.sh` file in the root of your Homestead directory (the same directory containing your `Homestead.yaml` file). Within this file, you may make any customization you would like; however, the `user-customizations.sh` should not be version controlled.

Provider Specific Settings

VirtualBox

`natdnshostresolver`

By default, Homestead configures the `natdnshostresolver` setting to `on`. This allows Homestead to use your host operating system's DNS settings. If you would like to override this behavior, add the following configuration options to your `Homestead.yaml` file:

```
provider: virtualbox
natdnshostresolver: 'off'
```

Symbolic Links On Windows

If symbolic links are not working properly on your Windows machine, you may need to add the following block to your `Vagrantfile`:

```
config.vm.provider "virtualbox" do |v|
    v.customize ["setextradata", :id, "VBoxInternal2/SharedFoldersEnableSymlinksCreate/v-root", "1"]
end
```

Packages

Laravel Horizon

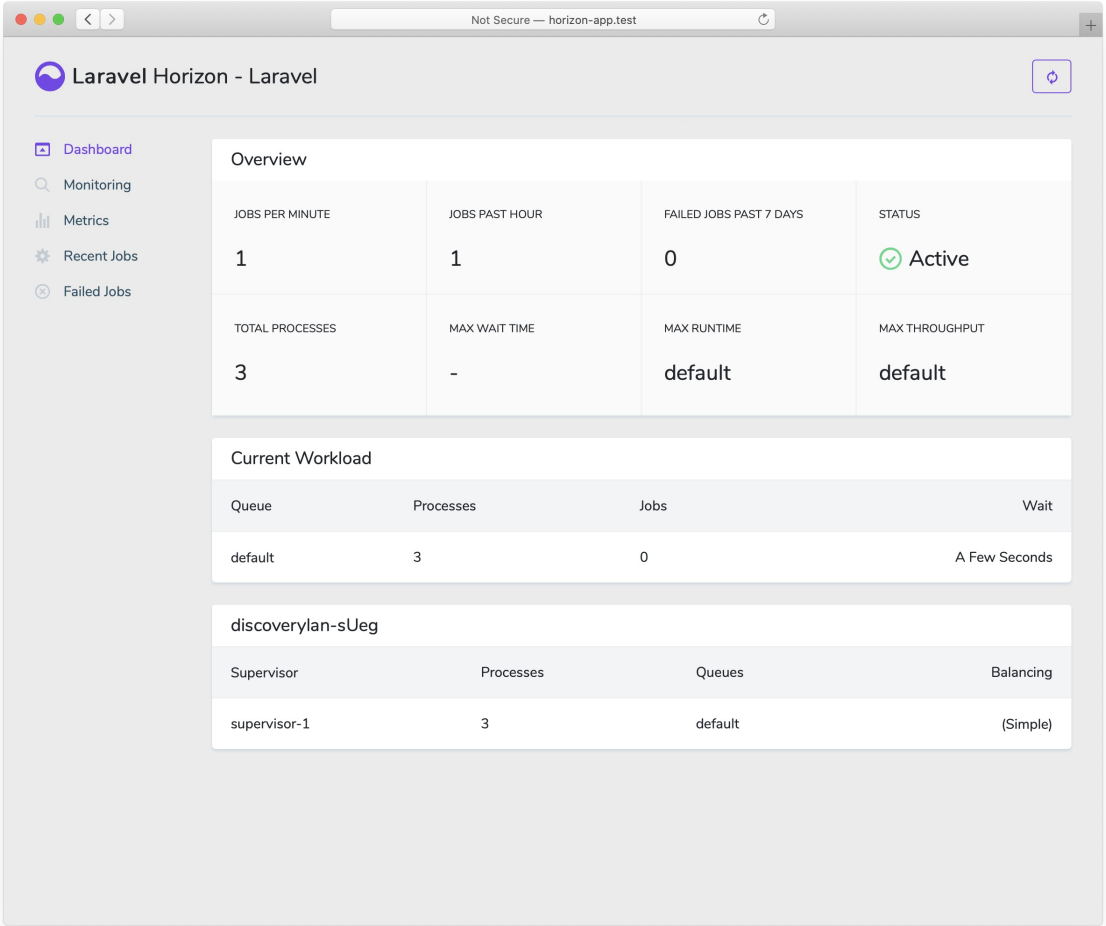
- Introduction
- Installation
  - Configuration
  - Balancing Strategies
  - Dashboard Authorization
- Upgrading Horizon
- Running Horizon
  - Deploying Horizon
- Tags
- Notifications
- Metrics
- Deleting Failed Jobs
- Clearing Jobs From Queues

Introduction

> TIP Before digging into Laravel Horizon, you should familiarize yourself with Laravel's base queue services. Horizon augments Laravel's queue with additional features that may be confusing if you are not already familiar with the basic queue features offered by Laravel.

Laravel Horizon provides a beautiful dashboard and code-driven configuration for your Laravel powered Redis queues. Horizon allows you to easily monitor key metrics of your queue system such as job throughput, runtime, and job failures.

When using Horizon, all of your queue worker configuration is stored in a single, simple configuration file. By defining your application's worker configuration in a version controlled file, you may easily scale or modify your application's queue workers when deploying your application.



Installation

> NOTE Laravel Horizon requires that you use Redis to power your queue. Therefore, you should ensure that your queue connection is set to `redis` in your application's `config/queue.php` configuration file.

You may install Horizon into your project using the Composer package manager:

```
composer require laravel/horizon
```

After installing Horizon, publish its assets using the `horizon:install` Artisan command:

```
php artisan horizon:install
```

Configuration

After publishing Horizon's assets, its primary configuration file will be located at `config/horizon.php`. This configuration file allows you to configure the queue worker options for your application. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.

Environments

After installation, the primary Horizon configuration option that you should familiarize yourself with is the `environments` configuration option. This configuration option is an array of environments that your application runs on and defines the worker process options for each environment. By default, this entry contains a `production` and `local` environment. However, you are free to add more environments as needed:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
        ],
    ],

    'local' => [
        'supervisor-1' => [
            'maxProcesses' => 3,
        ],
    ],
],
```

When you start Horizon, it will use the worker process configuration options for the environment that your application is running on. Typically, the environment is determined by the value of the `APP_ENV` environment variable. For example, the default `local` Horizon environment is configured to start three worker processes and automatically balance the number of worker processes assigned to each queue. The default `production` environment is configured to start a maximum of 10 worker processes and automatically balance the number of worker processes assigned to each queue.

> NOTE You should ensure that the `environments` portion of your `horizon` configuration file contains an entry for each environment on which you plan to run Horizon.

Supervisors

As you can see in Horizon's default configuration file. Each environment can contain one or more "supervisors". By default, the configuration file defines this supervisor as `supervisor-1`; however, you are free to name your supervisors whatever you want. Each supervisor is essentially responsible for "supervising" a group of worker processes and takes care of balancing worker processes across queues.

You may add additional supervisors to a given environment if you would like to define a new group of worker processes that should run in that environment. You may choose to do this if you would like to define a different balancing strategy or worker process count for a given queue used by your application.

Default Values

Within Horizon's default configuration file, you will notice a `defaults` configuration option. This configuration option specifies the default values for your application's supervisors. The supervisor's default configuration values will be merged into the supervisor's configuration for each environment, allowing you to avoid unnecessary repetition when defining your supervisors.

Balancing Strategies

Unlike Laravel's default queue system, Horizon allows you to choose from three worker balancing strategies: `simple`, `auto`, and `false`. The `simple` strategy, which is the configuration file's default, splits incoming jobs evenly between worker processes:

```
'balance' => 'simple',
```

The `auto` strategy adjusts the number of worker processes per queue based on the current workload of the queue. For example, if your `notifications` queue has 1,000 pending jobs while your `render` queue is empty, Horizon will allocate more workers to your `notifications` queue until the queue is empty.

When using the `auto` strategy, you may define the `minProcesses` and `maxProcesses` configuration options to control the minimum and maximum number of worker processes Horizon should scale up and down to:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default'],
            'balance' => 'auto',
            'minProcesses' => 1,
```

```
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
            'tries' => 3,
        ],
    ],
],
```

The `balanceMaxShift` and `balanceCooldown` configuration values to determine how quickly Horizon will scale to meet worker demand. In the example above, a maximum of one new process will be created or destroyed every three seconds. You are free to tweak these values as necessary based on your application's needs.

When the `balance` option is set to `false`, the default Laravel behavior will be used, which processes queues in the order they are listed in your configuration.

Dashboard Authorization

Horizon exposes a dashboard at the `/horizon` URI. By default, you will only be able to access this dashboard in the `local` environment. However, within your `app/Providers/HorizonServiceProvider.php` file, there is an authorization gate definition. This authorization gate controls access to Horizon in **non-local** environments. You are free to modify this gate as needed to restrict access to your Horizon installation:

```
/**
 * Register the Horizon gate.
 *
 * This gate determines who can access Horizon in non-local environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewHorizon', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

Alternative Authentication Strategies

Remember that Laravel automatically injects the authenticated user into the gate closure. If your application is providing Horizon security via another method, such as IP restrictions, then your Horizon users may not need to "login". Therefore, you will need to change `function ($user)` closure signature above to `function ($user = null)` in order to force Laravel to not require authentication.

Upgrading Horizon

When upgrading to a new major version of Horizon, it's important that you carefully review the upgrade guide. In addition, when upgrading to any new Horizon version, you should re-publish Horizon's assets:

```
php artisan horizon:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `horizon:publish` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan horizon:publish --ansi"
        ]
    }
}
```

Running Horizon

Once you have configured your supervisors and workers in your application's `config/horizon.php` configuration file, you may start Horizon using the `horizon` Artisan command. This single command will start all of the

configured worker processes for the current environment:

```
php artisan horizon
```

You may pause the Horizon process and instruct it to continue processing jobs using the `horizon:pause` and `horizon:continue` Artisan commands:

```
php artisan horizon:pause
```

```
php artisan horizon:continue
```

You may also pause and continue specific Horizon supervisors using the `horizon:pause-supervisor` and `horizon:continue-supervisor` Artisan commands:

```
php artisan horizon:pause-supervisor supervisor-1
```

```
php artisan horizon:continue-supervisor supervisor-1
```

You may check the current status of the Horizon process using the `horizon:status` Artisan command:

```
php artisan horizon:status
```

You may gracefully terminate the Horizon process using the `horizon:terminate` Artisan command. Any jobs that are currently being processed by will be completed and then Horizon will stop executing:

```
php artisan horizon:terminate
```

Deploying Horizon

When you're ready to deploy Horizon to your application's actual server, you should configure a process monitor to monitor the `php artisan horizon` command and restart it if it exits unexpectedly. Don't worry, we'll discuss how to install a process monitor below.

During your application's deployment process, you should instruct the Horizon process to terminate so that it will be restarted by your process monitor and receive your code changes:

```
php artisan horizon:terminate
```

Installing Supervisor

Supervisor is a process monitor for the Linux operating system and will automatically restart your `horizon` process if it stops executing. To install Supervisor on Ubuntu, you may use the following command. If you are not using Ubuntu, you can likely install Supervisor using your operating system's package manager:

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

> TIP If configuring Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your Laravel projects.

Supervisor Configuration

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

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/example.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/example.com/horizon.log
stopwaitsecs=3600
```

> NOTE You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the monitored processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start horizon
```

TIP For more information on running Supervisor, consult the Supervisor documentation.

Tags

Horizon allows you to assign “tags” to jobs, including mailables, broadcast events, notifications, and queued event listeners. In fact, Horizon will intelligently and automatically tag most jobs depending on the Eloquent models that are attached to the job. For example, take a look at the following job:

```
<?php

namespace App\Jobs;

use App\Models\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The video instance.
     *
     * @var \App\Models\Video
     */
    public $video;

    /**
     * Create a new job instance.
     *
     * @param  \App\Models\Video  $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }

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

If this job is queued with an `App\Models\Video` instance that has an `id` attribute of `1`, it will automatically receive the tag `App\Models\Video:1`. This is because Horizon will search the job's properties for any Eloquent models. If Eloquent models are found, Horizon will intelligently tag the job using the model's class name and primary key:

```
use App\Jobs\RenderVideo;
use App\Models\Video;

$video = Video::find(1);

RenderVideo::dispatch($video);
```

Manually Tagging Jobs

If you would like to manually define the tags for one of your queueable objects, you may define a `tags` method on the class:

```
class RenderVideo implements ShouldQueue
{
    /**
     * Get the tags that should be assigned to the job.
     *
     * @return array
     */
    public function tags()
    {
        return ['render', 'video:'.$this->video->id];
    }
}
```

Notifications

> NOTE When configuring Horizon to send Slack or SMS notifications, you should review the prerequisites for the relevant notification channel.

If you would like to be notified when one of your queues has a long wait time, you may use the `Horizon::routeMailNotificationsTo`, `Horizon::routeSlackNotificationsTo`, and `Horizon::routeSmsNotificationsTo` methods. You may call these methods from the `boot` method of your application's `App\Providers\HorizonServiceProvider`:

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

    Horizon::routeSmsNotificationsTo('15556667777');
    Horizon::routeMailNotificationsTo('example@example.com');
    Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
}
```

Configuring Notification Wait Time Thresholds

You may configure how many seconds are considered a "long wait" within your application's `config/horizon.php` configuration file. The `waits` configuration option within this file allows you to control the long wait threshold for each connection / queue combination:

```
'waits' => [
    'redis:default' => 60,
    'redis:critical,high' => 90,
],
```

Metrics

Horizon includes a metrics dashboard which provides information regarding your job and queue wait times and throughput. In order to populate this dashboard, you should configure Horizon's `snapshot` Artisan command to run every five minutes via your application's scheduler:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

Deleting Failed Jobs

If you would like to delete a failed job, you may use the `horizon:forget` command. The `horizon:forget` command accepts the ID of the failed job as its only argument:

```
php artisan horizon:forget 5
```

Clearing Jobs From Queues

If you would like to delete all jobs from your application's default queue, you may do so using the `horizon:clear` Artisan command:

```
php artisan horizon:clear
```

You may provide the `queue` option to delete jobs from a specific queue:

```
php artisan horizon:clear --queue=emails
```

Packages

Laravel Passport

- Introduction
  - Passport Or Sanctum?
- Installation
  - Deploying Passport
  - Migration Customization
  - Upgrading Passport
- Configuration
  - Client Secret Hashing
  - Token Lifetimes
  - Overriding Default Models
- Issuing Access Tokens
  - Managing Clients
  - Requesting Tokens
  - Refreshing Tokens
  - Revoking Tokens
  - Purging Tokens
- Authorization Code Grant with PKCE
  - Creating The Client
  - Requesting Tokens
- Password Grant Tokens
  - Creating A Password Grant Client
  - Requesting Tokens
  - Requesting All Scopes
  - Customizing The User Provider
  - Customizing The Username Field
  - Customizing The Password Validation
- Implicit Grant Tokens
- Client Credentials Grant Tokens
- Personal Access Tokens
  - Creating A Personal Access Client
  - Managing Personal Access Tokens
- Protecting Routes
  - Via Middleware
  - Passing The Access Token
- Token Scopes
  - Defining Scopes
  - Default Scope
  - Assigning Scopes To Tokens
  - Checking Scopes
- Consuming Your API With JavaScript
- Events
- Testing

Introduction

Laravel Passport provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the League OAuth2 server that is maintained by Andy Millington and Simon Hamp.

> NOTE This documentation assumes you are already familiar with OAuth2. If you do not know anything about OAuth2, consider familiarizing yourself with the general terminology and features of OAuth2 before continuing.

Passport Or Sanctum?

Before getting started, you may wish to determine if your application would be better served by Laravel Passport or Laravel Sanctum. If your application absolutely needs to support OAuth2, then you should use Laravel Passport.

However, if you are attempting to authenticate a single-page application, mobile application, or issue API tokens, you should use Laravel Sanctum. Laravel Sanctum does not support OAuth2; however, it provides a much simpler API authentication development experience.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Passport's service provider registers its own database migration directory, so you should migrate your database after installing the package. The Passport migrations will create the tables your application needs to store OAuth2 clients and access tokens:

```
php artisan migrate
```

Next, you should execute the `passport:install` Artisan command. This command will create the encryption keys needed to generate secure access tokens. In addition, the command will create "personal access" and "password grant" clients which will be used to generate access tokens:

```
php artisan passport:install
```

> TIP If you would like to use UUIDs as the primary key value of the Passport `Client` model instead of auto-incrementing integers, please install Passport using the `uuids` option.

After running the `passport:install` command, add the `Laravel\Passport\HasApiTokens` trait to your `App\Models\User` model. This trait will provide a few helper methods to your model which allow you to inspect the authenticated user's token and scopes:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

Next, you should call the `Passport::routes` method within the `boot` method of your `App\Providers\AuthServiceProvider`. This method will register the routes necessary to issue access tokens and revoke access tokens, clients, and personal access tokens:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;
use Laravel\Passport\Passport;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Models\Model' => 'App\Policies\ModelPolicy',
    ];
```

```
    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, in your application's `config/auth.php` configuration file, you should set the `driver` option of the `api` authentication guard to `passport`. This will instruct your application to use Passport's `TokenGuard` when authenticating incoming API requests:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Client UUIDs

You may also run the `passport:install` command with the `--uuids` option present. This option will instruct Passport that you would like to use UUIDs instead of auto-incrementing integers as the Passport `Client` model's primary key values. After running the `passport:install` command with the `--uuids` option, you will be given additional instructions regarding disabling Passport's default migrations:

```
php artisan passport:install --uuids
```

Deploying Passport

When deploying Passport to your application's servers for the first time, you will likely need to run the `passport:keys` command. This command generates the encryption keys Passport needs in order to generate access tokens. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

If necessary, you may define the path where Passport's keys should be loaded from. You may use the `Passport::loadKeysFrom` method to accomplish this. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::loadKeysFrom(__DIR__.'/../secrets/oauth');
}
```

Loading Keys From The Environment

Alternatively, you may publish Passport's configuration file using the `vendor:publish` Artisan command:

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

After the configuration file has been published, you may load your application's encryption keys by defining them as environment variables:

```
PASSPORT_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
<private key here>
-----END RSA PRIVATE KEY-----"

PASSPORT_PUBLIC_KEY="-----BEGIN PUBLIC KEY-----
<public key here>
-----END PUBLIC KEY-----"
```

Migration Customization

If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-migrations
```

Upgrading Passport

When upgrading to a new major version of Passport, it's important that you carefully review the upgrade guide.

Configuration

Client Secret Hashing

If you would like your client's secrets to be hashed when stored in your database, you should call the `Passport::hashClientSecrets` method in the `boot` method of your `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

Passport::hashClientSecrets();
```

Once enabled, all of your client secrets will only be displayable to the user immediately after they are created. Since the plain-text client secret value is never stored in the database, it is not possible to recover the secret's value if it is lost.

Token Lifetimes

By default, Passport issues long-lived access tokens that expire after one year. If you would like to configure a longer / shorter token lifetime, you may use the `tokensExpireIn`, `refreshTokensExpireIn`, and `personalAccessTokensExpireIn` methods. These methods should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(now()->addDays(15));
    Passport::refreshTokensExpireIn(now()->addDays(30));
    Passport::personalAccessTokensExpireIn(now()->addMonths(6));
}
```

> NOTE The `expires_at` columns on Passport's database tables are read-only and for display purposes only. When issuing tokens, Passport stores the expiration information within the signed and encrypted tokens. If you need to invalidate a token you should revoke it.

Overriding Default Models

You are free to extend the models used internally by Passport by defining your own model and extending the corresponding Passport model:

```
use Laravel\Passport\Client as PassportClient;

class Client extends PassportClient
{
    // ...
}
```

After defining your model, you may instruct Passport to use your custom model via the `Laravel\Passport\Passport` class. Typically, you should inform Passport about your custom models in the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessClient;
use App\Models\Passport\Token;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::useTokenModel(Token::class);
    Passport::useClientModel(Client::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::usePersonalAccessClientModel(PersonalAccessClient::class);
}
```

Issuing Access Tokens

Using OAuth2 via authorization codes is how most developers are familiar with OAuth2. When using authorization codes, a client application will redirect a user to your server where they will either approve or deny the request to issue an access token to the client.

Managing Clients

First, developers building applications that need to interact with your application's API will need to register their application with yours by creating a "client". Typically, this consists of providing the name of their application and a URL that your application can redirect to after users approve their request for authorization.

The `passport:client` Command

The simplest way to create a client is using the `passport:client` Artisan command. This command may be used to create your own clients for testing your OAuth2 functionality. When you run the `client` command, Passport will prompt you for more information about your client and will provide you with a client ID and secret:

```
php artisan passport:client
```

Redirect URLs

If you would like to allow multiple redirect URLs for your client, you may specify them using a comma-delimited list when prompted for the URL by the `passport:client` command. Any URLs which contain commas should be URL encoded:

```
http://example.com/callback,http://examplefoo.com/callback
```

JSON API

Since your application's users will not be able to utilize the `client` command, Passport provides a JSON API that you may use to create clients. This saves you the trouble of having to manually code controllers for creating, updating, and deleting clients.

However, you will need to pair Passport's JSON API with your own frontend to provide a dashboard for your users to manage their clients. Below, we'll review all of the API endpoints for managing clients. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

`GET /oauth/clients`

This route returns all of the clients for the authenticated user. This is primarily useful for listing all of the user's clients so that they may edit or delete them:

```
axios.get('/oauth/clients')
    .then(response => {
        console.log(response.data);
    });
```

`POST /oauth/clients`

This route is used to create new clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization.

When a client is created, it will be issued a client ID and client secret. These values will be used when requesting access tokens from your application. The client creation route will return the new client instance:

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`PUT /oauth/clients/{client-id}`

This route is used to update clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization. The route will return the updated client instance:

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`DELETE /oauth/clients/{client-id}`

This route is used to delete clients:

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        //
    });
```

Requesting Tokens

Redirecting For Authorization

Once a client has been created, developers may use their client ID and secret to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://passport-app.com/oauth/authorize?'.$query);
});
```

> TIP Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Approving The Request

When receiving authorization requests, Passport will automatically display a template to the user allowing them to approve or deny the authorization request. If they approve the request, they will be redirected back to the `redirect_uri` that was specified by the consuming application. The `redirect_uri` must match the `redirect` URL that was specified when the client was created.

If you would like to customize the authorization approval screen, you may publish Passport's views using the `vendor:publish` Artisan command. The published views will be placed in the `resources/views/vendor/passport` directory:

```
php artisan vendor:publish --tag=passport-views
```

Sometimes you may wish to skip the authorization prompt, such as when authorizing a first-party client. You may accomplish this by extending the `Client` model and defining a `skipsAuthorization` method. If `skipsAuthorization` returns `true` the client will be approved and the user will be redirected back to the `redirect_uri` immediately:

```
<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;

class Client extends BaseClient
{
    /**
     * Determine if the client should skip the authorization prompt.
     *
     * @return bool
     */
    public function skipsAuthorization()
    {
        return $this->firstParty();
    }
}
```

Converting Authorization Codes To Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should first verify the `state` parameter against the value that was stored prior to the redirect. If the

state parameter matches then the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request:

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

Route::get('/callback', function (Request $request) {
    $state = $request->session()->pull('state');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response = Http::asForm()->post('http://passport-app.com/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code' => $request->code,
    ]);

    return $response->json();
});
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

> TIP Like the `/oauth/authorize` route, the `/oauth/token` route is defined for you by the `Passport::routes` method. There is no need to manually define this route.

JSON API

Passport also includes a JSON API for managing authorized access tokens. You may pair this with your own frontend to offer your users a dashboard for managing access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints. The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application.

`GET /oauth/tokens`

This route returns all of the authorized access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they can revoke them:

```
axios.get('/oauth/tokens')
    .then(response => {
        console.log(response.data);
    });
```

`DELETE /oauth/tokens/{token-id}`

This route may be used to revoke authorized access tokens and their related refresh tokens:

```
axios.delete('/oauth/tokens/' + tokenId);
```

Refreshing Tokens

If your application issues short-lived access tokens, users will need to refresh their access tokens via the refresh token that was provided to them when the access token was issued:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.com/oauth/token', [
    'grant_type' => 'refresh_token',
    'refresh_token' => 'the-refresh-token',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => '',
]);
```

```
return $response->json();
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

Revoking Tokens

You may revoke a token by using the `revokeAccessToken` method on the `Laravel\Passport\TokenRepository`. You may revoke a token's refresh tokens using the `revokeRefreshTokensByAccessTokenId` method on the `Laravel\Passport\RefreshTokenRepository`. These classes may be resolved using Laravel's service container:

```
use Laravel\Passport\TokenRepository;
use Laravel\Passport\RefreshTokenRepository;

$tokenRepository = app(TokenRepository::class);
$refreshTokenRepository = app(RefreshTokenRepository::class);

// Revoke an access token...
$tokenRepository->revokeAccessToken($tokenId);

// Revoke all of the token's refresh tokens...
$refreshTokenRepository->revokeRefreshTokensByAccessTokenId($tokenId);
```

Purging Tokens

When tokens have been revoked or expired, you might want to purge them from the database. Passport's included `passport:purge` Artisan command can do this for you:

```
# Purge revoked and expired tokens and auth codes...
php artisan passport:purge

# Only purge revoked tokens and auth codes...
php artisan passport:purge --revoked

# Only purge expired tokens and auth codes...
php artisan passport:purge --expired
```

You may also configure a scheduled job in your application's `App\Console\Kernel` class to automatically prune your tokens on a schedule:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('passport:purge')->hourly();
}
```

Authorization Code Grant with PKCE

The Authorization Code grant with "Proof Key for Code Exchange" (PKCE) is a secure way to authenticate single page applications or native applications to access your API. This grant should be used when you can't guarantee that the client secret will be stored confidentially or in order to mitigate the threat of having the authorization code intercepted by an attacker. A combination of a "code verifier" and a "code challenge" replaces the client secret when exchanging the authorization code for an access token.

Creating The Client

Before your application can issue tokens via the authorization code grant with PKCE, you will need to create a PKCE-enabled client. You may do this using the `passport:client` Artisan command with the `--public` option:

```
php artisan passport:client --public
```

Requesting Tokens

Code Verifier & Code Challenge

As this authorization grant does not provide a client secret, developers will need to generate a combination of a code verifier and a code challenge in order to request a token.

The code verifier should be a random string of between 43 and 128 characters containing letters, numbers, and `"-"`, `"."`, `"_"`, `"~"` characters, as defined in the RFC 7636 specification.

The code challenge should be a Base64 encoded string with URL and filename-safe characters. The trailing `'='` characters should be removed and no line breaks, whitespace, or other additional characters should be present.

```
$encoded = base64_encode(hash('sha256', $code_verifier, true));

$codeChallenge = strtr(rtrim($encoded, '='), '+/', '-_');
```

Redirecting For Authorization

Once a client has been created, you may use the client ID and the generated code verifier and code challenge to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route:

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $request->session()->put(
        'code_verifier', $code_verifier = Str::random(128)
    );

    $codeChallenge = strtr(rtrim(
        base64_encode(hash('sha256', $code_verifier, true))
    , '='), '+/', '-_');

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        'code_challenge' => $codeChallenge,
        'code_challenge_method' => 'S256',
    ]);

    return redirect('http://passport-app.com/oauth/authorize?'.$query);
});
```

Converting Authorization Codes To Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should verify the `state` parameter against the value that was stored prior to the redirect, as in the standard Authorization Code Grant.

If the state parameter matches, the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request along with the originally generated code verifier:

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

Route::get('/callback', function (Request $request) {
    $state = $request->session()->pull('state');

    $codeVerifier = $request->session()->pull('code_verifier');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );
```

```
    $response = Http::asForm()->post('http://passport-app.com/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code_verifier' => $codeVerifier,
        'code' => $request->code,
    ]);

    return $response->json();
});
```

Password Grant Tokens

The OAuth2 password grant allows your other first-party clients, such as a mobile application, to obtain an access token using an email address / username and password. This allows you to issue access tokens securely to your first-party clients without requiring your users to go through the entire OAuth2 authorization code redirect flow.

Creating A Password Grant Client

Before your application can issue tokens via the password grant, you will need to create a password grant client. You may do this using the `passport:client` Artisan command with the `--password` option. **If you have already run the `passport:install` command, you do not need to run this command:**

```
php artisan passport:client --password
```

Requesting Tokens

Once you have created a password grant client, you may request an access token by issuing a `POST` request to the `/oauth/token` route with the user's email address and password. Remember, this route is already registered by the `Passport::routes` method so there is no need to define it manually. If the request is successful, you will receive an `access_token` and `refresh_token` in the JSON response from the server:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.com/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '',
]);

return $response->json();
```

> TIP Remember, access tokens are long-lived by default. However, you are free to configure your maximum access token lifetime if needed.

Requesting All Scopes

When using the password grant or client credentials grant, you may wish to authorize the token for all of the scopes supported by your application. You can do this by requesting the `*` scope. If you request the `*` scope, the `can` method on the token instance will always return `true`. This scope may only be assigned to a token that is issued using the `password` or `client_credentials` grant:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.com/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '*',
]);
```

Customizing The User Provider

If your application uses more than one authentication user provider, you may specify which user provider the password grant client uses by providing a `--provider` option when creating the client via the `artisan passport:client --password` command. The given provider name should match a valid provider defined in your application's `config/auth.php` configuration file. You can then protect your route using middleware to ensure that only users from the guard's specified provider are authorized.

Customizing The Username Field

When authenticating using the password grant, Passport will use the `email` attribute of your authenticatable model as the "username". However, you may customize this behavior by defining a `findForPassport` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Find the user instance for the given username.
     *
     * @param  string  $username
     * @return \App\Models\User
     */
    public function findForPassport($username)
    {
        return $this->where('username', $username)->first();
    }
}
```

Customizing The Password Validation

When authenticating using the password grant, Passport will use the `password` attribute of your model to validate the given password. If your model does not have a `password` attribute or you wish to customize the password validation logic, you can define a `validateForPassportPasswordGrant` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Hash;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Validate the password of the user for the Passport password grant.
     *
     * @param  string  $password
     * @return bool
     */
    public function validateForPassportPasswordGrant($password)
    {
        return Hash::check($password, $this->password);
    }
}
```

Implicit Grant Tokens

The implicit grant is similar to the authorization code grant; however, the token is returned to the client without exchanging an authorization code. This grant is most commonly used for JavaScript or mobile applications where the client credentials can't be securely stored. To enable the grant, call the `enableImplicitGrant` method in the `boot` method of your applications's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Once the grant has been enabled, developers may use their client ID to request an access token from your application. The consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
use Illuminate\Http\Request;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'token',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://passport-app.com/oauth/authorize?'.$query);
});
```

> TIP Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Client Credentials Grant Tokens

The client credentials grant is suitable for machine-to-machine authentication. For example, you might use this grant in a scheduled job which is performing maintenance tasks over an API.

Before your application can issue tokens via the client credentials grant, you will need to create a client credentials grant client. You may do this using the `--client` option of the `passport:client` Artisan command:

```
php artisan passport:client --client
```

Next, to use this grant type, you need to add the `CheckClientCredentials` middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

Then, attach the middleware to a route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client');
```

To restrict access to the route to specific scopes, you may provide a comma-delimited list of the required scopes when attaching the `client` middleware to the route:

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

```
    ...
})->middleware('client:check-status,your-scope');
```

Retrieving Tokens

To retrieve a token using this grant type, make a request to the `oauth/token` endpoint:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.com/oauth/token', [
    'grant_type' => 'client_credentials',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => 'your-scope',
]);

return $response->json()['access_token'];
```

Personal Access Tokens

Sometimes, your users may want to issue access tokens to themselves without going through the typical authorization code redirect flow. Allowing users to issue tokens to themselves via your application's UI can be useful for allowing users to experiment with your API or may serve as a simpler approach to issuing access tokens in general.

Creating A Personal Access Client

Before your application can issue personal access tokens, you will need to create a personal access client. You may do this by executing the `passport:client` Artisan command with the `--personal` option. If you have already run the `passport:install` command, you do not need to run this command:

```
php artisan passport:client --personal
```

After creating your personal access client, place the client's ID and plain-text secret value in your application's `.env` file:

```
PASSPORT_PERSONAL_ACCESS_CLIENT_ID="client-id-value"
PASSPORT_PERSONAL_ACCESS_CLIENT_SECRET="unhashed-client-secret-value"
```

Managing Personal Access Tokens

Once you have created a personal access client, you may issue tokens for a given user using the `createToken` method on the `App\Models\User` model instance. The `createToken` method accepts the name of the token as its first argument and an optional array of scopes as its second argument:

```
use App\Models\User;

$user = User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport also includes a JSON API for managing personal access tokens. You may pair this with your own frontend to offer your users a dashboard for managing personal access tokens. Below, we'll review all of the API endpoints for managing personal access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

GET /oauth/scopes

This route returns all of the scopes defined for your application. You may use this route to list the scopes a user may assign to a personal access token:

```
axios.get('/oauth/scopes')
    .then(response => {
        console.log(response.data);
    });
```

GET /oauth/personal-access-tokens

This route returns all of the personal access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they may edit or revoke them:

```
axios.get('/oauth/personal-access-tokens')
    .then(response => {
        console.log(response.data);
    });
```

POST /oauth/personal-access-tokens

This route creates new personal access tokens. It requires two pieces of data: the token's `name` and the `scopes` that should be assigned to the token:

```
const data = {
    name: 'Token Name',
    scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
    .then(response => {
        console.log(response.data.accessToken);
    })
    .catch (response => {
        // List errors on response...
    });
```

DELETE /oauth/personal-access-tokens/{token-id}

This route may be used to revoke personal access tokens:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protecting Routes

Via Middleware

Passport includes an authentication guard that will validate access tokens on incoming requests. Once you have configured the `api` guard to use the `passport` driver, you only need to specify the `auth:api` middleware on any routes that should require a valid access token:

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

Multiple Authentication Guards

If your application authenticates different types of users that perhaps use entirely different Eloquent models, you will likely need to define a guard configuration for each user provider type in your application. This allows you to protect requests intended for specific user providers. For example, given the following guard configuration the `config/auth.php` configuration file:

```
'api' => [
    'driver' => 'passport',
    'provider' => 'users',
],
```

```
'api-customers' => [
    'driver' => 'passport',
    'provider' => 'customers',
],
```

The following route will utilize the `api-customers` guard, which uses the `customers` user provider, to authenticate incoming requests:

```
Route::get('/customer', function () {
    //
})->middleware('auth:api-customers');
```

> TIP For more information on using multiple user providers with Passport, please consult the password grant documentation.

Passing The Access Token

When calling routes that are protected by Passport, your application's API consumers should specify their access token as a `Bearer` token in the `Authorization` header of their request. For example, when using the Guzzle HTTP library:

```
use Illuminate\Support\Facades\Http;

$response = Http::withHeaders([
    'Accept' => 'application/json',
    'Authorization' => 'Bearer '.$accessToken,
])->get('https://passport-app.com/api/user');

return $response->json();
```

Token Scopes

Scopes allow your API clients to request a specific set of permissions when requesting authorization to access an account. For example, if you are building an e-commerce application, not all API consumers will need the ability to place orders. Instead, you may allow the consumers to only request authorization to access order shipment statuses. In other words, scopes allow your application's users to limit the actions a third-party application can perform on their behalf.

Defining Scopes

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your application's `App\Providers\AuthServiceProvider` class. The `tokensCan` method accepts an array of scope names and scope descriptions. The scope description may be anything you wish and will be displayed to users on the authorization approval screen:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensCan([
        'place-orders' => 'Place orders',
        'check-status' => 'Check order status',
    ]);
}
```

Default Scope

If a client does not request any specific scopes, you may configure your Passport server to attach default scope(s) to the token using the `setDefaultScope` method. Typically, you should call this method from the `boot`

method of your application's `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);

Passport::setDefaultScope([
    'check-status',
    'place-orders',
]);
```

Assigning Scopes To Tokens

When Requesting Authorization Codes

When requesting an access token using the authorization code grant, consumers should specify their desired scopes as the `scope` query string parameter. The `scope` parameter should be a space-delimited list of scopes:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://passport-app.com/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `App\Models\User` model's `createToken` method, you may pass the array of desired scopes as the second argument to the method:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Checking Scopes

Passport includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given scope. To get started, add the following middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Check For All Scopes

The `scopes` middleware may be assigned to a route to verify that the incoming request's access token has all of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
})->middleware(['auth:api', 'scopes:check-status,place-orders']);
```

Check For Any Scopes

The `scope` middleware may be assigned to a route to verify that the incoming request's access token has *at least one* of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
})->middleware(['auth:api', 'scope:check-status,place-orders']);
```

Checking Scopes On A Token Instance

Once an access token authenticated request has entered your application, you may still check if the token has a given scope using the `tokenCan` method on the authenticated `App\Models\User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        //
    }
});
```

Additional Scope Methods

The `scopeIds` method will return an array of all defined IDs / names:

```
use Laravel\Passport\Passport;

Passport::scopeIds();
```

The `scopes` method will return an array of all defined scopes as instances of `Laravel\Passport\Scope`:

```
Passport::scopes();
```

The `scopesFor` method will return an array of `Laravel\Passport\Scope` instances matching the given IDs / names:

```
Passport::scopesFor(['place-orders', 'check-status']);
```

You may determine if a given scope has been defined using the `hasScope` method:

```
Passport::hasScope('place-orders');
```

Consuming Your API With JavaScript

When building an API, it can be extremely useful to be able to consume your own API from your JavaScript application. This approach to API development allows your own application to consume the same API that you are sharing with the world. The same API may be consumed by your web application, mobile applications, third-party applications, and any SDKs that you may publish on various package managers.

Typically, if you want to consume your API from your JavaScript application, you would need to manually send an access token to the application and pass it with each request to your application. However, Passport includes a middleware that can handle this for you. All you need to do is add the `CreateFreshApiToken` middleware to your `web` middleware group in your `app/Http/Kernel.php` file:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

> NOTE You should ensure that the `CreateFreshApiToken` middleware is the last middleware listed in your middleware stack.

This middleware will attach a `laravel_token` cookie to your outgoing responses. This cookie contains an encrypted JWT that Passport will use to authenticate API requests from your JavaScript application. The JWT has a lifetime equal to your `session.lifetime` configuration value. Now, since the browser will automatically send the cookie with all subsequent requests, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/api/user')
    .then(response => {
        console.log(response.data);
    });
```

Customizing The Cookie Name

If needed, you can customize the `laravel_token` cookie's name using the `Passport::cookie` method. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider`

class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::cookie('custom_name');
}
```

CSRF Protection

When using this method of authentication, you will need to ensure a valid CSRF token header is included in your requests. The default Laravel JavaScript scaffolding includes an Axios instance, which will automatically use the encrypted `XSRF-TOKEN` cookie value to send a `X-XSRF-TOKEN` header on same-origin requests.

> TIP If you choose to send the `X-CSRF-TOKEN` header instead of `X-XSRF-TOKEN`, you will need to use the unencrypted token provided by `csrf_token()`.

Events

Passport raises events when issuing access tokens and refresh tokens. You may use these events to prune or revoke other access tokens in your database. If you would like, you may attach listeners to these events in your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],

    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

Testing

Passport's `actingAs` method may be used to specify the currently authenticated user as well as its scopes. The first argument given to the `actingAs` method is the user instance and the second is an array of scopes that should be granted to the user's token:

```
use App\Models\User;
use Laravel\Passport\Passport;

public function test_servers_can_be_created()
{
    Passport::actingAs(
        User::factory()->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(201);
}
```

Passport's `actingAsClient` method may be used to specify the currently authenticated client as well as its scopes. The first argument given to the `actingAsClient` method is the client instance and the second is an array of scopes that should be granted to the client's token:

```
use Laravel\Passport\Client;
use Laravel\Passport\Passport;

public function test_orders_can_be_retrieved()
{
    Passport::actingAsClient(
        Client::factory()->create(),
        ['check-status']
    );

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

    $response->assertStatus(200);
}
```

Packages

Laravel Sail

- Introduction
- Installation & Setup
  - Installing Sail Into Existing Applications
  - Configuring A Bash Alias
- Starting & Stopping Sail
- Executing Commands
  - Executing PHP Commands
  - Executing Composer Commands
  - Executing Artisan Commands
  - Executing Node / NPM Commands
- Interacting With Databases
  - MySQL
  - Redis
- Running Tests
  - Laravel Dusk
- Previewing Emails
- Container CLI
- PHP Versions
- Sharing Your Site
- Customization

Introduction

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

At its heart, Sail is the `docker-compose.yml` file and the `sail` script that is stored at the root of your project. The `sail` script provides a CLI with convenient methods for interacting with the Docker containers defined by the `docker-compose.yml` file.

Laravel Sail is supported on macOS, Linux, and Windows (via WSL2).

Installation & Setup

Laravel Sail is automatically installed with all new Laravel applications so you may start using it immediately. To learn how to create a new Laravel application, please consult Laravel's installation documentation for your operating system.

Installing Sail Into Existing Applications

If you are interested in using Sail with an existing Laravel application, you may simply install Sail using the Composer package manager. Of course, these steps assume that your existing local development environment allows you to install Composer dependencies:

```
composer require laravel/sail --dev
```

After Sail has been installed, you may run the `sail:install` Artisan command. This command will publish Sail's `docker-compose.yml` file to the root of your application:

```
php artisan sail:install
```

Finally, you may start Sail. To continue learning how to use Sail, please continue reading the remainder of this documentation:

```
./vendor/bin/sail up
```

Configuring A Bash Alias

By default, Sail commands are invoked using the `vendor/bin/sail` script that is included with all new Laravel applications:

```
./vendor/bin/sail up
```

However, instead of repeatedly typing `vendor/bin/sail` to execute Sail commands, you may wish to configure a Bash alias that allows you to execute Sail's commands more easily:

```
alias sail='bash vendor/bin/sail'
```

Once the Bash alias has been configured, you may execute Sail commands by simply typing `sail`. The remainder of this documentation's examples will assume that you have configured this alias:

```
sail up
```

Starting & Stopping Sail

Laravel Sail's `docker-compose.yml` file defines a Docker variety of containers that work together to help you build Laravel applications. Each of these containers is an entry within the `services` configuration of your `docker-compose.yml` file. The `laravel.test` container is the primary application container that will be serving your application.

Before starting Sail, you should ensure that no other web servers or databases are running on your local computer. To start all of the Docker containers defined in your application's `docker-compose.yml` file, you should execute the `up` command:

```
sail up
```

To start all of the Docker containers in the background, you may start Sail in "detached" mode:

```
sail up -d
```

Once the application's containers have been started, you may access the project in your web browser at: http://localhost.

To stop all of the containers, you may simply press Control + C to stop the container's execution. Or, if the containers are running in the background, you may use the `down` command:

```
sail down
```

Executing Commands

When using Laravel Sail, your application is executing within a Docker container and is isolated from your local computer. However, Sail provides a convenient way to run various commands against your application such as arbitrary PHP commands, Artisan commands, Composer commands, and Node / NPM commands.

When reading the Laravel documentation, you will often see references to Composer, Artisan, and Node / NPM commands that do not reference Sail. Those examples assume that these tools are installed on your local computer. If you are using Sail for your local Laravel development environment, you should execute those commands using Sail:

```
# Running Artisan commands locally...
php artisan queue:work

# Running Artisan commands within Laravel Sail...
sail artisan queue:work
```

Executing PHP Commands

PHP commands may be executed using the `php` command. Of course, these commands will execute using the PHP version that is configured for your application. To learn more about the PHP versions available to Laravel

Sail, consult the PHP version documentation:

```
sail php --version
```

```
sail php script.php
```

Executing Composer Commands

Composer commands may be executed using the `composer` command. Laravel Sail's application container includes a Composer 2.x installation:

```
sail composer require laravel/sanctum
```

Installing Composer Dependencies For Existing Applications

If you are developing an application with a team, you may not be the one that initially creates the Laravel application. Therefore, none of the application's Composer dependencies, including Sail, will be installed after you clone the application's repository to your local computer.

You may install the application's dependencies by navigating to the application's directory and executing the following command. This command uses a small Docker container containing PHP and Composer to install the application's dependencies:

```
docker run --rm \
    -v $(pwd):/opt \
    -w /opt \
    laravelsail/php80-composer:latest \
    composer install
```

Executing Artisan Commands

Laravel Artisan commands may be executed using the `artisan` command:

```
sail artisan queue:work
```

Executing Node / NPM Commands

Node commands may be executed using the `node` command while NPM commands may be executed using the `npm` command:

```
sail node --version
```

```
sail npm run prod
```

Interacting With Databases

MySQL

As you may have noticed, your application's `docker-compose.yml` file contains an entry for a MySQL container. This container uses a Docker volume so that the data stored in your database is persisted even when stopping and restarting your containers. In addition, when the MySQL container is starting, it will ensure a database exists whose name matches the value of your `DB_DATABASE` environment variable.

Once you have started your containers, you may connect to the MySQL instance within your application by setting your `DB_HOST` environment variable within your application's `.env` file to `mysql`.

To connect to your application's MySQL database from your local machine, you may use a graphical database management application such as TablePlus. By default, the MySQL database is accessible at `localhost` port 3306.

Redis

Your application's `docker-compose.yml` file also contains an entry for a Redis container. This container uses a Docker volume so that the data stored in your Redis data is persisted even when stopping and restarting your containers. Once you have started your containers, you may connect to the Redis instance within your application by setting your `REDIS_HOST` environment variable within your application's `.env` file to `redis`.

To connect to your application's Redis database from your local machine, you may use a graphical database management application such as TablePlus. By default, the Redis database is accessible at `localhost` port 6379.

Running Tests

Laravel provides amazing testing support out of the box, and you may use Sail's `test` command to run your applications feature and unit tests. Any CLI options that are accepted by PHPUnit may also be passed to the `test` command:

```
sail test
```

```
sail test --group orders
```

The Sail `test` command is equivalent to running the `test` Artisan command:

```
sail artisan test
```

Laravel Dusk

Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. Thanks to Sail, you may run these tests without ever installing Selenium or other tools on your local computer. To get started, uncomment the Selenium service in your application's `docker-compose.yml` file:

```
selenium:
    image: 'selenium/standalone-chrome'
    volumes:
        - '/dev/shm:/dev/shm'
    networks:
        - sail
```

Next, ensure that the `laravel.test` service in your application's `docker-compose.yml` file has a `depends_on` entry for `selenium`:

```
depends_on:
    - mysql
    - redis
    - selenium
```

Finally, you may run your Dusk test suite by starting Sail and running the `dusk` command:

```
sail dusk
```

Previewing Emails

Laravel Sail's default `docker-compose.yml` file contains a service entry for MailHog. MailHog intercepts emails sent by your application during local development and provides a convenient web interface so that you can preview your email messages in your browser. When using Sail, MailHog's default host is `mailhog` and is available via port 1025:

```
MAIL_HOST=mailhog
MAIL_PORT=1025
```

When Sail is running, you may access the MailHog web interface at: http://localhost:8025

Container CLI

Sometimes you may wish to start a Bash session within your application's container. You may use the `shell` command to connect to your application's container, allowing you to inspect its files and installed services as well execute arbitrary shell commands within the container:

```
sail shell
```

To start a new Laravel Tinker session, you may execute the `tinker` command:

```
sail tinker
```

PHP Versions

Sail currently supports serving your application via PHP 8.0 or PHP 7.4. To change the PHP version that is used to serve your application, you should update the `build` definition of the `laravel.test` container in your application's `docker-compose.yml` file:

```
# PHP 8.0
context: ./vendor/laravel/sail/runtimes/8.0

# PHP 7.4
context: ./vendor/laravel/sail/runtimes/7.4
```

In addition, you may wish to update your `image` name to reflect the version of PHP being used by your application. This option is also defined in your application's `docker-compose.yml` file:

```
image: sail-8.0/app
```

After updating your application's `docker-compose.yml` file, you should rebuild your container images:

```
sail build --no-cache
```

```
sail up
```

Sharing Your Site

Sometimes you may need to share your site publicly in order to preview your site for a colleague or to test webhook integrations with your application. To share your site, you may use the `share` command. After executing this command, you will be issued a random `laravel-sail.site` URL that you may use to access your application:

```
sail share
```

If you would like to choose the subdomain for your shared site, you may provide the `subdomain` option when executing the `share` command:

```
sail share --subdomain=my-sail-site
```

> TIP The `share` command is powered by Expose, an open source tunneling service by BeyondCode.

Customization

Since Sail is just Docker, you are free to customize nearly everything about it. To publish Sail's own Dockerfiles, you may execute the `sail:publish` command:

```
php artisan sail:publish
```

After running this command, the Dockerfiles and other configuration files used by Laravel Sail will be placed within a `docker` directory in your application's root directory. After customizing your Sail installation, you may rebuild your application's containers using the `build` command:

```
sail build --no-cache
```

Packages

Laravel Sanctum

- Introduction
  - How It Works
- Installation
- Configuration
  - Overriding Default Models
- API Token Authentication
  - Issuing API Tokens
  - Token Abilities
  - Protecting Routes
  - Revoking Tokens
- SPA Authentication
  - Configuration
  - Authenticating
  - Protecting Routes
  - Authorizing Private Broadcast Channels
- Mobile Application Authentication
  - Issuing API Tokens
  - Protecting Routes
  - Revoking Tokens
- Testing

Introduction

Laravel Sanctum provides a featherweight authentication system for SPAs (single page applications), mobile applications, and simple, token based APIs. Sanctum allows each user of your application to generate multiple API tokens for their account. These tokens may be granted abilities / scopes which specify which actions the tokens are allowed to perform.

How It Works

Laravel Sanctum exists to solve two separate problems. Let's discuss each before digging deeper into the library.

API Tokens

First, Sanctum is a simple package you may use to issue API tokens to your users without the complication of OAuth. This feature is inspired by GitHub and other applications which issue "personal access tokens". For example, imagine the "account settings" of your application has a screen where a user may generate an API token for their account. You may use Sanctum to generate and manage those tokens. These tokens typically have a very long expiration time (years), but may be manually revoked by the user at anytime.

Laravel Sanctum offers this feature by storing user API tokens in a single database table and authenticating incoming HTTP requests via the `Authorization` header which should contain a valid API token.

SPA Authentication

Second, Sanctum exists to offer a simple way to authenticate single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository, such as a SPA created using Vue CLI or a Next.js application.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. Typically, Sanctum utilizes Laravel's `web` authentication guard to accomplish this. This provides the benefits of CSRF protection, session authentication, as well as protects

against leakage of the authentication credentials via XSS.

Sanctum will only attempt to authenticate using cookies when the incoming request originates from your own SPA frontend. When Sanctum examines an incoming HTTP request, it will first check for an authentication cookie and, if none is present, Sanctum will then examine the `Authorization` header for a valid API token.

> TIP It is perfectly fine to use Sanctum only for API token authentication or only for SPA authentication. Just because you use Sanctum does not mean you are required to use both features it offers.

Installation

You may install Laravel Sanctum via the Composer package manager:

```
composer require laravel/sanctum
```

Next, you should publish the Sanctum configuration and migration files using the `vendor:publish` Artisan command. The `sanctum` configuration file will be placed in your application's `config` directory:

```
php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"
```

Finally, you should run your database migrations. Sanctum will create one database table in which to store API tokens:

```
php artisan migrate
```

Next, if you plan to utilize Sanctum to authenticate an SPA, you should add Sanctum's middleware to your `api` middleware group within your application's `app/Http/Kernel.php` file:

```
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

Migration Customization

If you are not going to use Sanctum's default migrations, you should call the `Sanctum::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations by executing the following command: `php artisan vendor:publish --tag=sanctum-migrations`

Configuration

Overriding Default Models

Although not typically required, you are free to extend the `PersonalAccessToken` model used internally by Sanctum:

```
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;

class PersonalAccessToken extends SanctumPersonalAccessToken
{
    // ...
}
```

Then, you may instruct Sanctum to use your custom model via the `usePersonalAccessTokenModel` method provided by Sanctum. Typically, you should call this method in the `boot` method of one of your application's service providers:

```
use App\Models\Passport\PersonalAccessToken;
use Laravel\Sanctum\Sanctum;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
```

```
public function boot()
{
    Sanctum::usePersonalAccessTokenModel(PersonalAccessToken::class);
}
```

API Token Authentication

> TIP You should not use API tokens to authenticate your own first-party SPA. Instead, use Sanctum's built-in SPA authentication features.

Issuing API Tokens

Sanctum allows you to issue API tokens / personal access tokens that may be used to authenticate API requests to your application. When making requests using API tokens, the token should be included in the `Authorization` header as a `Bearer` token.

To begin issuing tokens for users, your User model should use the `Laravel\Sanctum\HasApiTokens` trait:

```
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

To issue a token, you may use the `createToken` method. The `createToken` method returns a `Laravel\Sanctum\NewAccessToken` instance. API tokens are hashed using SHA-256 hashing before being stored in your database, but you may access the plain-text value of the token using the `plainTextToken` property of the `NewAccessToken` instance. You should display this value to the user immediately after the token has been created:

```
use Illuminate\Http\Request;

Route::post('/tokens/create', function (Request $request) {
    $token = $request->user()->createToken($request->token_name);

    return ['token' => $token->plainTextToken];
});
```

You may access all of the user's tokens using the `tokens` Eloquent relationship provided by the `HasApiTokens` trait:

```
foreach ($user->tokens as $token) {
    //
}
```

Token Abilities

Sanctum allows you to assign "abilities" to tokens. Abilities serve a similar purpose as OAuth's "scopes". You may pass an array of string abilities as the second argument to the `createToken` method:

```
return $user->createToken('token-name', ['server:update'])->plainTextToken;
```

When handling an incoming request authenticated by Sanctum, you may determine if the token has a given ability using the `tokenCan` method:

```
if ($user->tokenCan('server:update')) {
    //
}
```

First-Party UI Initiated Requests

For convenience, the `tokenCan` method will always return `true` if the incoming authenticated request was from your first-party SPA and you are using Sanctum's built-in SPA authentication.

However, this does not necessarily mean that your application has to allow the user to perform the action. Typically, your application's authorization policies will determine if the token has been granted the permission

to perform the abilities as well as check that the user instance itself should be allowed to perform the action.

For example, if we imagine an application that manages servers, this might mean checking that token is authorized to update servers **and** that the server belongs to the user:

```
return $request->user()->id === $server->user_id &&
       $request->user()->tokenCan('server:update')
```

At first, allowing the `tokenCan` method to be called and always return `true` for first-party UI initiated requests may seem strange; however, it is convenient to be able to always assume an API token is available and can be inspected via the `tokenCan` method. By taking this approach, you may always call the `tokenCan` method within your application's authorizations policies without worrying about whether the request was triggered from your application's UI or was initiated by one of your API's third-party consumers.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your protected routes within your `routes/web.php` and `routes/api.php` route files. This guard will ensure that incoming requests are authenticated as either stateful, cookie authenticated requests or contain a valid API token header if the request is from a third party.

You may be wondering why we suggest that you authenticate the routes within your application's `routes/web.php` file using the `sanctum` guard. Remember, Sanctum will first attempt to authenticate incoming requests using Laravel's typical session authentication cookie. If that cookie is not present then Sanctum will attempt to authenticate the request using a token in the request's `Authorization` header. In addition, authenticating all requests using Sanctum ensures that we may always call the `tokenCan` method on the currently authenticated user instance:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Revoking Tokens

You may "revoke" tokens by deleting them from your database using the `tokens` relationship that is provided by the `Laravel\Sanctum\HasApiTokens` trait:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke the token that was used to authenticate the current request...
$request->user()->currentAccessToken()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

SPA Authentication

Sanctum also exists to provide a simple method of authenticating single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. This approach to authentication provides the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.

> NOTE In order to authenticate, your SPA and API must share the same top-level domain. However, they may be placed on different subdomains.

Configuration

Configuring Your First-Party Domains

First, you should configure which domains your SPA will be making requests from. You may configure these domains using the `stateful` configuration option in your `sanctum` configuration file. This configuration setting determines which domains will maintain "stateful" authentication using Laravel session cookies when making requests to your API.

> NOTE If you are accessing your application via a URL that includes a port (`127.0.0.1:8000`), you should ensure that you include the port number with the domain.

Sanctum Middleware

Next, you should add Sanctum's middleware to your `api` middleware group within your `app/Http/Kernel.php` file. This middleware is responsible for ensuring that incoming requests from your SPA can authenticate using Laravel's session cookies, while still allowing requests from third parties or mobile applications to authenticate using API tokens:

```
use Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful;

'api' => [
    EnsureFrontendRequestsAreStateful::class,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

CORS & Cookies

If you are having trouble authenticating with your application from an SPA that executes on a separate subdomain, you have likely misconfigured your CORS (Cross-Origin Resource Sharing) or session cookie settings.

You should ensure that your application's CORS configuration is returning the `Access-Control-Allow-Credentials` header with a value of `True`. This may be accomplished by setting the `supports_credentials` option within your application's `config/cors.php` configuration file to `true`.

In addition, you should enable the `withCredentials` option on your application's global `axios` instance. Typically, this should be performed in your `resources/js/bootstrap.js` file. If you are not using Axios to make HTTP requests from your frontend, you should perform the equivalent configuration on your own HTTP client:

```
axios.defaults.withCredentials = true;
```

Finally, you should ensure your application's session cookie domain configuration supports any subdomain of your root domain. You may accomplish this by prefixing the domain with a leading `.` within your application's `config/session.php` configuration file:

```
'domain' => '.domain.com',
```

Authenticating

CSRF Protection

To authenticate your SPA, your SPA's "login" page should first make a request to the `/sanctum/csrf-cookie` endpoint to initialize CSRF protection for the application:

```
axios.get('/sanctum/csrf-cookie').then(response => {
    // Login...
});
```

During this request Laravel will set an `XSRF-TOKEN` cookie containing the current CSRF token. This token should then be passed in an `X-XSRF-TOKEN` header on subsequent requests, which some HTTP client libraries like Axios and the Angular HttpClient will do automatically for you. If your JavaScript HTTP library does not set the value for you, you will need to manually set the `X-XSRF-TOKEN` header to match the value of the `XSRF-TOKEN` cookie that is set by this route.

Logging In

Once CSRF protection has been initialized, you should make a `POST` request to the your Laravel application's `/login` route. This `/login` route may be implemented manually or using a headless authentication package like Laravel Fortify.

If the login request is successful, you will be authenticated and subsequent requests to your application's routes will automatically be authenticated via the session cookie that the Laravel application issued to your client. In addition, since your application already made a request to the `/sanctum/csrf-cookie` route, subsequent requests should automatically receive CSRF protection as long as your JavaScript HTTP client sends the value of the `XSRF-TOKEN` cookie in the `X-XSRF-TOKEN` header.

Of course, if your user's session expires due to lack of activity, subsequent requests to the Laravel application may receive 401 or 419 HTTP error response. In this case, you should redirect the user to your SPA's login page.

> NOTE You are free to write your own `/login` endpoint; however, you should ensure that it authenticates the user using the standard, session based authentication services that Laravel provides. Typically, this means using the `web` authentication guard.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your API routes within your `routes/api.php` file. This guard will ensure that incoming requests are authenticated as either a stateful authenticated requests from your SPA or contain a valid API token header if the request is from a third party:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Authorizing Private Broadcast Channels

If your SPA needs to authenticate with private / presence broadcast channels, you should place the `Broadcast::routes` method call within your `routes/api.php` file:

```
Broadcast::routes(['middleware' => ['auth:sanctum']]);
```

Next, in order for Pusher's authorization requests to succeed, you will need to provide a custom Pusher `authorizer` when initializing Laravel Echo. This allows your application to configure Pusher to use the `axios` instance that is properly configured for cross-domain requests:

```
window.Echo = new Echo({
    broadcaster: "pusher",
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    encrypted: true,
    key: process.env.MIX_PUSHER_APP_KEY,
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(false, response.data);
                })
                .catch(error => {
                    callback(true, error);
                });
            }
        };
    },
})
```

Mobile Application Authentication

You may also use Sanctum tokens to authenticate your mobile application's requests to your API. The process for authenticating mobile application requests is similar to authenticating third-party API requests; however, there are small differences in how you will issue the API tokens.

Issuing API Tokens

To get started, create a route that accepts the user's email / username, password, and device name, then exchanges those credentials for a new Sanctum token. The "device name" given to this endpoint is for informational purposes and may be any value you wish. In general, the device name value should be a name the user would recognize, such as "Nuno's iPhone 12".

Typically, you will make a request to the token endpoint from your mobile application's "login" screen. The endpoint will return the plain-text API token which may then be stored on the mobile device and used to make additional API requests:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

Route::post('/sanctum/token', function (Request $request) {
    $request->validate([
        'email' => 'required|email',
        'password' => 'required',
        'device_name' => 'required',
    ]);

    $user = User::where('email', $request->email)->first();

    if (! $user || ! Hash::check($request->password, $user->password)) {
        throw ValidationException::withMessages([
            'email' => ['The provided credentials are incorrect.'],
        ]);
    }

    return $user->createToken($request->device_name)->plainTextToken;
});
```

When the mobile application uses the token to make an API request to your application, it should pass the token in the `Authorization` header as a `Bearer` token.

TIP When issuing tokens for a mobile application, you are also free to specify token abilities

Protecting Routes

As previously documented, you may protect routes so that all incoming requests must be authenticated by attaching the `sanctum` authentication guard to the routes:

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Revoking Tokens

To allow users to revoke API tokens issued to mobile devices, you may list them by name, along with a "Revoke" button, within an "account settings" portion of your web application's UI. When the user clicks the "Revoke" button, you can delete the token from the database. Remember, you can access a user's API tokens via the `tokens` relationship provided by the `Laravel\Sanctum\HasApiTokens` trait:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

Testing

While testing, the `Sanctum::actingAs` method may be used to authenticate a user and specify which abilities should be granted to their token:

```
use App\Models\User;
use Laravel\Sanctum\Sanctum;

public function test_task_list_can_be_retrieved()
{
    Sanctum::actingAs(
        User::factory()->create(),
        ['view-tasks']
    );

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

    $response->assertOk();
}
```

If you would like to grant all abilities to the token, you should include `*` in the ability list provided to the `actingAs` method:

```
Sanctum::actingAs(
    User::factory()->create(),
    ['*']
);
```

Packages

Laravel Scout

- Introduction
- Installation
  - Driver Prerequisites
  - Queueing
- Configuration
  - Configuring Model Indexes
  - Configuring Searchable Data
  - Configuring The Model ID
  - Identifying Users
- Indexing
  - Batch Import
  - Adding Records
  - Updating Records
  - Removing Records
  - Pausing Indexing
  - Conditionally Searchable Model Instances
- Searching
  - Where Clauses
  - Pagination
  - Soft Deleting
  - Customizing Engine Searches
- Custom Engines
- Builder Macros

Introduction

Laravel Scout provides a simple, driver based solution for adding full-text search to your Eloquent models. Using model observers, Scout will automatically keep your search indexes in sync with your Eloquent records.

Currently, Scout ships with an Algolia driver; however, writing custom drivers is simple and you are free to extend Scout with your own search implementations.

Installation

First, install Scout via the Composer package manager:

```
composer require laravel/scout
```

After installing Scout, you should publish the Scout configuration file using the `vendor:publish` Artisan command. This command will publish the `scout.php` configuration file to your application's `config` directory:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

Finally, add the `Laravel\Scout\Searchable` trait to the model you would like to make searchable. This trait will register a model observer that will automatically keep the model in sync with your search driver:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;
}
```

Driver Prerequisites

Algolia

When using the Algolia driver, you should configure your Algolia `id` and `secret` credentials in your `config/scout.php` configuration file. Once your credentials have been configured, you will also need to install the Algolia PHP SDK via the Composer package manager:

```
composer require algolia/algoliasearch-client-php
```

Queueing

While not strictly required to use Scout, you should strongly consider configuring a queue driver before using the library. Running a queue worker will allow Scout to queue all operations that sync your model information to your search indexes, providing much better response times for your application's web interface.

Once you have configured a queue driver, set the value of the `queue` option in your `config/scout.php` configuration file to `true`:

```
'queue' => true,
```

Configuration

Configuring Model Indexes

Each Eloquent model is synced with a given search "index", which contains all of the searchable records for that model. In other words, you can think of each index like a MySQL table. By default, each model will be persisted to an index matching the model's typical "table" name. Typically, this is the plural form of the model name; however, you are free to customize the model's index by overriding the `searchableAs` method on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the name of the index associated with the model.
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

Configuring Searchable Data

By default, the entire `toArray` form of a given model will be persisted to its search index. If you would like to customize the data that is synchronized to the search index, you may override the `toSearchableArray` method on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
```

```
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize the data array...

        return $array;
    }
}
```

Configuring The Model ID

By default, Scout will use the primary key of the model as model's unique ID / key that is stored in the search index. If you need to customize this behavior, you may override the `getScoutKey` and the `getScoutKeyName` methods on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * Get the value used to index the model.
     *
     * @return mixed
     */
    public function getScoutKey()
    {
        return $this->email;
    }

    /**
     * Get the key name used to index the model.
     *
     * @return mixed
     */
    public function getScoutKeyName()
    {
        return 'email';
    }
}
```

Identifying Users

Scout also allows you to auto identify users when using Algolia. Associating the authenticated user with search operations may be helpful when viewing your search analytics within Algolia's dashboard. You can enable user identification by defining a `SCOUT_IDENTIFY` environment variable as `true` in your application's `.env` file:

```
SCOUT_IDENTIFY=true
```

Enabling this feature this will also pass the request's IP address and your authenticated user's primary identifier to Algolia so this data is associated with any search request that is made by the user.

Indexing

Batch Import

If you are installing Scout into an existing project, you may already have database records you need to import into your indexes. Scout provides a `scout:import` Artisan command that you may use to import all of your existing records into your search indexes:

```
php artisan scout:import "App\Models\Post"
```

The `flush` command may be used to remove all of a model's records from your search indexes:

```
php artisan scout:flush "App\Models\Post"
```

Modifying The Import Query

If you would like to modify the query that is used to retrieve all of your models for batch importing, you may define a `makeAllSearchableUsing` method on your model. This is a great place to add any eager relationship loading that may be necessary before importing your models:

```
/**
 * Modify the query used to retrieve models when making all of the models searchable.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $query
 * @return \Illuminate\Database\Eloquent\Builder
 */
protected function makeAllSearchableUsing($query)
{
    return $query->with('author');
}
```

Adding Records

Once you have added the `Laravel\Scout\Searchable` trait to a model, all you need to do is `save` or `create` a model instance and it will automatically be added to your search index. If you have configured Scout to use queues this operation will be performed in the background by your queue worker:

```
use App\Models\Order;

$order = new Order;

// ...

$order->save();
```

Adding Records Via Query

If you would like to add a collection of models to your search index via an Eloquent query, you may chain the `searchable` method onto the Eloquent query. The `searchable` method will chunk the results of the query and add the records to your search index. Again, if you have configured Scout to use queues, all of the chunks will be imported in the background by your queue workers:

```
use App\Models\Order;

Order::where('price', '>', 100)->searchable();
```

You may also call the `searchable` method on an Eloquent relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to add the model instances to their corresponding index:

```
$orders->searchable();
```

> TIP The `searchable` method can be considered an "upsert" operation. In other words, if the model record is already in your index, it will be updated. If it does not exist in the search index, it will be added to the index.

Updating Records

To update a searchable model, you only need to update the model instance's properties and `save` the model to your database. Scout will automatically persist the changes to your search index:

```
use App\Models\Order;

$order = Order::find(1);

// Update the order...

$order->save();
```

You may also invoke the `searchable` method on an Eloquent query instance to update a collection of models. If the models do not exist in your search index, they will be created:

```
Order::where('price', '>', 100)->searchable();
```

If you would like to update the search index records for all of the models in a relationship, you may invoke the `searchable` on the relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to update the model instances in their corresponding index:

```
$orders->searchable();
```

Removing Records

To remove a record from your index you may simply `delete` the model from the database. This may be done even if you are using soft deleted models:

```
use App\Models\Order;

$order = Order::find(1);

$order->delete();
```

If you do not want to retrieve the model before deleting the record, you may use the `unsearchable` method on an Eloquent query instance:

```
Order::where('price', '>', 100)->unsearchable();
```

If you would like to remove the search index records for all of the models in a relationship, you may invoke the `unsearchable` on the relationship instance:

```
$user->orders()->unsearchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `unsearchable` method on the collection instance to remove the model instances from their corresponding index:

```
$orders->unsearchable();
```

Pausing Indexing

Sometimes you may need to perform a batch of Eloquent operations on a model without syncing the model data to your search index. You may do this using the `withoutSyncingToSearch` method. This method accepts a single closure which will be immediately executed. Any model operations that occur within the closure will not be synced to the model's index:

```
use App\Models\Order;

Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

Conditionally Searchable Model Instances

Sometimes you may need to only make a model searchable under certain conditions. For example, imagine you have `App\Models\Post` model that may be in one of two states: "draft" and "published". You may only want to allow "published" posts to be searchable. To accomplish this, you may define a `shouldBeSearchable` method on your model:

```
/**
 * Determine if the model should be searchable.
 *
 * @return bool
 */
public function shouldBeSearchable()
{
    return $this->isPublished();
}
```

The `shouldBeSearchable` method is only applied when manipulating models through the `save` and `create` methods, queries, or relationships. Directly making models or collections searchable using the `searchable` method will override the result of the `shouldBeSearchable` method.

Searching

You may begin searching a model using the `search` method. The search method accepts a single string that will be used to search your models. You should then chain the `get` method onto the search query to retrieve the Eloquent models that match the given search query:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->get();
```

Since Scout searches return a collection of Eloquent models, you may even return the results directly from a route or controller and they will automatically be converted to JSON:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return Order::search($request->search)->get();
});
```

If you would like to get the raw search results before they are converted to Eloquent models, you may use the `raw` method:

```
$orders = Order::search('Star Trek')->raw();
```

Custom Indexes

Search queries will typically be performed on the index specified by the model's `searchableAs` method. However, you may use the `within` method to specify a custom index that should be searched instead:

```
$orders = Order::search('Star Trek')
    ->within('tv_shows_popularity_desc')
    ->get();
```

Where Clauses

Scout allows you to add simple "where" clauses to your search queries. Currently, these clauses only support basic numeric equality checks and are primarily useful for scoping search queries by an owner ID. Since a search index is not a relational database, more advanced "where" clauses are not currently supported:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->where('user_id', 1)->get();
```

Pagination

In addition to retrieving a collection of models, you may paginate your search results using the `paginate`

method. This method will return an `Illuminate\Pagination\LengthAwarePaginator` instance just as if you had paginated a traditional Eloquent query:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->paginate();
```

You may specify how many models to retrieve per page by passing the amount as the first argument to the `paginate` method:

```
$orders = Order::search('Star Trek')->paginate(15);
```

Once you have retrieved the results, you may display the results and render the page links using Blade just as if you had paginated a traditional Eloquent query:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

Of course, if you would like to retrieve the pagination results as JSON, you may return the paginator instance directly from a route or controller:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    return Order::search($request->input('query'))->paginate(15);
});
```

Soft Deleting

If your indexed models are soft deleting and you need to search your soft deleted models, set the `soft_delete` option of the `config/scout.php` configuration file to `true`:

```
'soft_delete' => true,
```

When this configuration option is `true`, Scout will not remove soft deleted models from the search index. Instead, it will set a hidden `__soft_deleted` attribute on the indexed record. Then, you may use the `withTrashed` or `onlyTrashed` methods to retrieve the soft deleted records when searching:

```
use App\Models\Order;

// Include trashed records when retrieving results...
$orders = Order::search('Star Trek')->withTrashed()->get();

// Only include trashed records when retrieving results...
$orders = Order::search('Star Trek')->onlyTrashed()->get();
```

> TIP When a soft deleted model is permanently deleted using `forceDelete`, Scout will remove it from the search index automatically.

Customizing Engine Searches

If you need to perform advanced customization of the search behavior of an engine you may pass a closure as the second argument to the `search` method. For example, you could use this callback to add geo-location data to your search options before the search query is passed to Algolia:

```
use Algolia\AlgoliaSearch\SearchIndex;
use App\Models\Order;

Order::search(
    'Star Trek',
    function (SearchIndex $algolia, string $query, array $options) {
        $options['body']['query']['bool']['filter']['geo_distance'] = [
            'distance' => '1000km',
            'location' => ['lat' => 36, 'lon' => 111],
```

```
        ];

        return $algolia->search($query, $options);
    }
)->get();
```

Custom Engines

Writing The Engine

If one of the built-in Scout search engines doesn't fit your needs, you may write your own custom engine and register it with Scout. Your engine should extend the `Laravel\Scout\Engines\Engine` abstract class. This abstract class contains eight methods your custom engine must implement:

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

You may find it helpful to review the implementations of these methods on the `Laravel\Scout\Engines\AlgoliaEngine` class. This class will provide you with a good starting point for learning how to implement each of these methods in your own engine.

Registering The Engine

Once you have written your custom engine, you may register it with Scout using the `extend` method of the Scout engine manager. Scout's engine manager may be resolved from the Laravel service container. You should call the `extend` method from the `boot` method of your `App\Providers\AppServiceProvider` class or any other service provider used by your application:

```
use App\ScoutExtensions\MySqlSearchEngine
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

Once your engine has been registered, you may specify it as your default Scout `driver` in your application's `config/scout.php` configuration file:

```
'driver' => 'mysql',
```

Builder Macros

If you would like to define a custom Scout search builder method, you may use the `macro` method on the `Laravel\Scout\Builder` class. Typically, "macros" should be defined within a service provider's `boot` method:

```
use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;
use Laravel\Scout\Builder;

/**
 * Bootstrap any application services.
 *
 * @return void
```

```
 */
public function boot()
{
    Builder::macro('count', function () {
        return $this->engine()->getTotalCount(
            $this->engine()->search($this)
        );
    });
}
```

The `macro` function accepts a macro name as its first argument and a closure as its second argument. The macro's closure will be executed when calling the macro name from a `Laravel\Scout\Builder` implementation:

```
use App\Models\Order;

Order::search('Star Trek')->count();
```

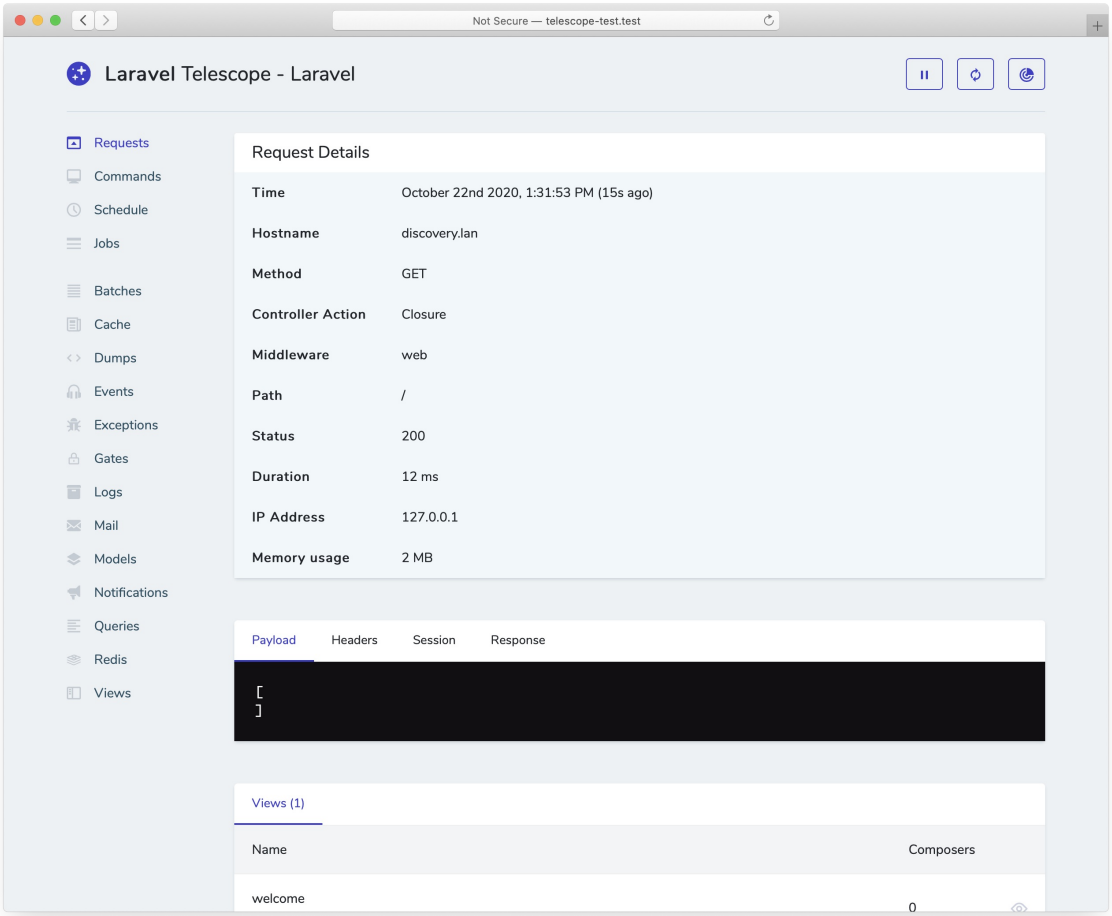
Packages

Laravel Telescope

- Introduction
- Installation
  - Local Only Installation
  - Configuration
  - Data Pruning
  - Dashboard Authorization
- Upgrading Telescope
- Filtering
  - Entries
  - Batches
- Tagging
- Available Watchers
  - Batch Watcher
  - Cache Watcher
  - Command Watcher
  - Dump Watcher
  - Event Watcher
  - Exception Watcher
  - Gate Watcher
  - Job Watcher
  - Log Watcher
  - Mail Watcher
  - Model Watcher
  - Notification Watcher
  - Query Watcher
  - Redis Watcher
  - Request Watcher
  - Schedule Watcher
  - View Watcher
- Displaying User Avatars

Introduction

Telescope makes a wonderful companion to your local Laravel development environment. Telescope provides insight into the requests coming into your application, exceptions, log entries, database queries, queued jobs, mail, notifications, cache operations, scheduled tasks, variable dumps, and more.



Installation

You may use the Composer package manager to install Telescope into your Laravel project:

```
composer require laravel/telescope
```

After installing Telescope, publish its assets using the `telescope:install` Artisan command. After installing Telescope, you should also run the `migrate` command in order to create the tables needed to store Telescope's data:

```
php artisan telescope:install

php artisan migrate
```

Migration Customization

If you are not going to use Telescope's default migrations, you should call the `Telescope::ignoreMigrations` method in the `register` method of your application's `App\Providers\AppServiceProvider` class. You may export the default migrations using the following command: `php artisan vendor:publish --tag=telescope-migrations`

Local Only Installation

If you plan to only use Telescope to assist your local development, you may install Telescope using the `--dev` flag:

```
composer require laravel/telescope --dev

php artisan telescope:install
```

```
php artisan migrate
```

After running `telescope:install`, you should remove the `TelescopeServiceProvider` service provider registration from your application's `config/app.php` configuration file. Instead, manually register Telescope's service providers in the `register` method of your `App\Providers\AppServiceProvider` class. We will ensure the current environment is `local` before registering the providers:

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    if ($this->app->environment('local')) {
        $this->app->register(\Laravel\Telescope\TelescopeServiceProvider::class);
        $this->app->register(TelescopeServiceProvider::class);
    }
}
```

Finally, you should also prevent the Telescope package from being auto-discovered by adding the following to your `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "laravel/telescope"
        ]
    }
},
```

Configuration

After publishing Telescope's assets, its primary configuration file will be located at `config/telescope.php`. This configuration file allows you to configure your watcher options. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.

If desired, you may disable Telescope's data collection entirely using the `enabled` configuration option:

```
'enabled' => env('TELESCOPE_ENABLED', true),
```

Data Pruning

Without pruning, the `telescope_entries` table can accumulate records very quickly. To mitigate this, you should schedule the `telescope:prune` Artisan command to run daily:

```
$schedule->command('telescope:prune')->daily();
```

By default, all entries older than 24 hours will be pruned. You may use the `hours` option when calling the command to determine how long to retain Telescope data. For example, the following command will delete all records created over 48 hours ago:

```
$schedule->command('telescope:prune --hours=48')->daily();
```

Dashboard Authorization

The Telescope dashboard may be accessed at the `/telescope` route. By default, you will only be able to access this dashboard in the `local` environment. Within your `app/Providers/TelescopeServiceProvider.php` file, there is an authorization gate definition. This authorization gate controls access to Telescope in **non-local** environments. You are free to modify this gate as needed to restrict access to your Telescope installation:

```
/**
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope in non-local environments.
 *
 * @return void
```

```
 */
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

NOTE You should ensure you change your `APP_ENV` environment variable to `production` in your production environment. Otherwise, your Telescope installation will be publicly available.

Upgrading Telescope

When upgrading to a new major version of Telescope, it's important that you carefully review the upgrade guide.

In addition, when upgrading to any new Telescope version, you should re-publish Telescope's assets:

```
php artisan telescope:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `telescope:publish` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan telescope:publish --ansi"
        ]
    }
}
```

Filtering

Entries

You may filter the data that is recorded by Telescope via the `filter` closure that is defined in your `App\Providers\TelescopeServiceProvider` class. By default, this closure records all data in the `local` environment and exceptions, failed jobs, scheduled tasks, and data with monitored tags in all other environments:

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filter(function (IncomingEntry $entry) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entry->isReportableException() ||
            $entry->isFailedJob() ||
            $entry->isScheduledTask() ||
            $entry->hasMonitoredTag();
    });
}
```

Batches

While the `filter` closure filters data for individual entries, you may use the `filterBatch` method to register a closure that filters all data for a given request or console command. If the closure returns `true`, all of the entries

are recorded by Telescope:

```
use Illuminate\Support\Collection;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filterBatch(function (Collection $entries) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entries->contains(function ($entry) {
            return $entry->isReportableException() ||
                $entry->isFailedJob() ||
                $entry->isScheduledTask() ||
                $entry->hasMonitoredTag();
            });
    });
}
```

Tagging

Telescope allows you to search entries by "tag". Often, tags are Eloquent model class names or authenticated user IDs which Telescope automatically adds to entries. Occasionally, you may want to attach your own custom tags to entries. To accomplish this, you may use the `Telescope::tag` method. The `tag` method accepts a closure which should return an array of tags. The tags returned by the closure will be merged with any tags Telescope would automatically attach to the entry. Typically, you should call the `tag` method within the `register` method of your `App\Providers\TelescopeServiceProvider` class:

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::tag(function (IncomingEntry $entry) {
        return $entry->type === 'request'
                    ? ['status:'.$entry->content['response_status']]
                    : [];
    });
 }
```

Available Watchers

Telescope "watchers" gather application data when a request or console command is executed. You may customize the list of watchers that you would like to enable within your `config/telescope.php` configuration file:

```
'watchers' => [
    Watchers\CacheWatcher::class => true,
    Watchers\CommandWatcher::class => true,
    ...
],
```

Some watchers also allow you to provide additional customization options:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
```

```
        'slow' => 100,
    ],
    ...
],
```

Batch Watcher

The batch watcher records information about queued batches, including the job and connection information.

Cache Watcher

The cache watcher records data when a cache key is hit, missed, updated and forgotten.

Command Watcher

The command watcher records the arguments, options, exit code, and output whenever an Artisan command is executed. If you would like to exclude certain commands from being recorded by the watcher, you may specify the command in the `ignore` option within your `config/telescope.php` file:

```
'watchers' => [
    Watchers\CommandWatcher::class => [
        'enabled' => env('TELESCOPE_COMMAND_WATCHER', true),
        'ignore' => ['key:generate'],
    ],
    ...
],
```

Dump Watcher

The dump watcher records and displays your variable dumps in Telescope. When using Laravel, variables may be dumped using the global `dump` function. The dump watcher tab must be open in a browser for the dump to be recorded, otherwise the dumps will be ignored by the watcher.

Event Watcher

The event watcher records the payload, listeners, and broadcast data for any events dispatched by your application. The Laravel framework's internal events are ignored by the Event watcher.

Exception Watcher

The exception watcher records the data and stack trace for any reportable exceptions that are thrown by your application.

Gate Watcher

The gate watcher records the data and result of gate and policy checks by your application. If you would like to exclude certain abilities from being recorded by the watcher, you may specify those in the `ignore_abilities` option in your `config/telescope.php` file:

```
'watchers' => [
    Watchers\GateWatcher::class => [
        'enabled' => env('TELESCOPE_GATE_WATCHER', true),
        'ignore_abilities' => ['viewNova'],
    ],
    ...
],
```

Job Watcher

The job watcher records the data and status of any jobs dispatched by your application.

Log Watcher

The log watcher records the log data for any logs written by your application.

Mail Watcher

The mail watcher allows you to view an in-browser preview of emails sent by your application along with their associated data. You may also download the email as an `.eml` file.

Model Watcher

The model watcher records model changes whenever an Eloquent model event is dispatched. You may specify which model events should be recorded via the watcher's `events` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
    ],
    ...
],
```

If you would like to record the number of models hydrated during a given request, enable the `hydrations` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
        'hydrations' => true,
    ],
    ...
],
```

Notification Watcher

The notification watcher records all notifications sent by your application. If the notification triggers an email and you have the mail watcher enabled, the email will also be available for preview on the mail watcher screen.

Query Watcher

The query watcher records the raw SQL, bindings, and execution time for all queries that are executed by your application. The watcher also tags any queries slower than 100 milliseconds as `slow`. You may customize the slow query threshold using the watcher's `slow` option:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 50,
    ],
    ...
],
```

Redis Watcher

The Redis watcher records all Redis commands executed by your application. If you are using Redis for caching, cache commands will also be recorded by the Redis watcher.

Request Watcher

The request watcher records the request, headers, session, and response data associated with any requests handled by the application. You may limit your recorded response data via the `size_limit` (in kilobytes) option:

```
'watchers' => [
    Watchers\RequestWatcher::class => [
        'enabled' => env('TELESCOPE_REQUEST_WATCHER', true),
        'size_limit' => env('TELESCOPE_RESPONSE_SIZE_LIMIT', 64),
```

```
    ],
    ...
],
```

Schedule Watcher

The schedule watcher records the command and output of any scheduled tasks run by your application.

View Watcher

The view watcher records the view name, path, data, and "composers" used when rendering views.

Displaying User Avatars

The Telescope dashboard displays the user avatar for the user that was authenticated when a given entry was saved. By default, Telescope will retrieve avatars using the Gravatar web service. However, you may customize the avatar URL by registering a callback in your `App\Providers\TelescopeServiceProvider` class. The callback will receive the user's ID and email address and should return the user's avatar image URL:

```
use App\Models\User;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    // ...

    Telescope::avatar(function ($id, $email) {
        return '/avatars/'.User::find($id)->avatar_path;
    });
}
```

Packages

Laravel Valet

- Introduction
- Installation
  - Upgrading Valet
- Serving Sites
  - The "Park" Command
  - The "Link" Command
  - Securing Sites With TLS
- Sharing Sites
  - Sharing Sites Via Ngrok
  - Sharing Sites Via Expose
  - Sharing Sites On Your Local Network
- Site Specific Environment Variables
- Proxying Services
- Custom Valet Drivers
  - Local Drivers
- Other Valet Commands
- Valet Directories & Files

Introduction

Valet is a Laravel development environment for macOS minimalists. Laravel Valet configures your Mac to always run Nginx in the background when your machine starts. Then, using DnsMasq, Valet proxies all requests on the `*.test` domain to point to sites installed on your local machine.

In other words, Valet is a blazing fast Laravel development environment that uses roughly 7 MB of RAM. Valet isn't a complete replacement for Sail or Homestead, but provides a great alternative if you want flexible basics, prefer extreme speed, or are working on a machine with a limited amount of RAM.

Out of the box, Valet support includes, but is not limited to:

- Laravel
- Lumen
- Bedrock
- CakePHP 3
- Concrete5
- Contao
- Craft
- Drupal
- ExpressionEngine
- Jigsaw
- Joomla
- Katana
- Kirby
- Magento
- OctoberCMS
- Sculpin
- Slim
- Statamic
- Static HTML
- Symfony
- WordPress
- Zend

However, you may extend Valet with your own custom drivers.

Installation

> NOTE Valet requires macOS and Homebrew. Before installation, you should make sure that no other programs such as Apache or Nginx are binding to your local machine's port 80.

To get started, you first need to ensure that Homebrew is up to date using the `update` command:

```
brew update
```

Next, you should use Homebrew to install PHP:

```
brew install php
```

After installing PHP, you are ready to install the Composer package manager. In addition, you should make sure the `~/.composer/vendor/bin` directory is in your system's "PATH". After Composer has been installed, you may install Laravel Valet as a global Composer package:

```
composer global require laravel/valet
```

Finally, you may execute Valet's `install` command. This will configure and install Valet and DnsMasq. In addition, the Valet daemon will be configured to launch when your system starts:

```
valet install
```

Once Valet is installed, try pinging any `*.test` domain on your terminal using a command such as `ping foobar.test`. If Valet is installed correctly you should see this domain responding on `127.0.0.1`.

Valet will automatically start its daemon each time your machine boots. There is no need to run `valet start` or `valet install` ever again once the initial Valet installation is complete.

PHP Versions

Valet allows you to switch PHP versions using the `valet use php@version` command. Valet will install the specified PHP version via Homebrew if it is not already installed:

```
valet use php@7.2
```

```
valet use php
```

> NOTE Valet only serves one PHP version at a time, even if you have multiple PHP versions installed.

Database

If your application needs a database, check out DBngin. DBngin provides a free, all-in-one database management tool that includes MySQL, PostgreSQL, and Redis. After DBngin has been installed, you can connect to your database at `127.0.0.1` using the `root` username and an empty string for the password.

Resetting Your Installation

If you are having trouble getting your Valet installation to run properly, executing the `composer global update` command followed by `valet install` will reset your installation and can solve a variety of problems. In rare cases it may be necessary to "hard reset" Valet by executing `valet uninstall --force` followed by `valet install`.

Upgrading Valet

You may update your Valet installation by executing the `composer global update` command in your terminal. After upgrading, it is good practice to run the `valet install` command so Valet can make additional upgrades to your configuration files if necessary.

Serving Sites

Once Valet is installed, you're ready to start serving your Laravel applications. Valet provides two commands to help you serve your applications: `park` and `link`.

The `park` Command

The `park` command registers a directory on your machine that contains your applications. Once the directory has been "parked" with Valet, all of the directories within that directory will be accessible in your web browser at `http://<directory-name>.test`:

```
cd ~/Sites

valet park
```

That's all there is to it. Now, any application you create within your "parked" directory will automatically be served using the `http://<directory-name>.test` convention. So, if your parked directory contains a directory named "laravel", the application within that directory will be accessible at `http://laravel.test`.

The `link` Command

The `link` command can also be used to serve your Laravel applications. This command is useful if you want to serve a single site in a directory and not the entire directory:

```
cd ~/Sites/laravel

valet link
```

Once an application has been linked to Valet using the `link` command, you may access the application using its directory name. So, the site that was linked in the example above may be accessed at `http://laravel.test`.

If you would like to serve the application at a different hostname, you may pass the hostname to the `link` command. For example, you may run the following command to make an application available at `http://application.test`:

```
cd ~/Sites/laravel

valet link application
```

You may execute the `links` command to display a list of all of your linked directories:

```
valet links
```

The `unlink` command may be used to destroy the symbolic link for a site:

```
cd ~/Sites/laravel

valet unlink
```

Securing Sites With TLS

By default, Valet serves sites over HTTP. However, if you would like to serve a site over encrypted TLS using HTTP/2, you may use the `secure` command. For example, if your site is being served by Valet on the `laravel.test` domain, you should run the following command to secure it:

```
valet secure laravel
```

To "unsecure" a site and revert back to serving its traffic over plain HTTP, use the `unsecure` command. Like the `secure` command, this command accepts the hostname that you wish to unsecure:

```
valet unsecure laravel
```

Sharing Sites

Valet even includes a command to share your local sites with the world, providing an easy way to test your site on mobile devices or share it with team members and clients.

Sharing Sites Via Ngrok

To share a site, navigate to the site's directory in your terminal and run Valet's `share` command. A publicly accessible URL will be inserted into your clipboard and is ready to paste directly into your browser or share with your team:

```
cd ~/Sites/laravel

valet share
```

To stop sharing your site, you may press `Control + C`.

> TIP You may pass additional Ngrok parameters to the share command, such as `valet share --region=eu`. For more information, consult the ngrok documentation.

Sharing Sites Via Expose

If you have Expose installed, you can share your site by navigating to the site's directory in your terminal and running the `expose` command. Consult the Expose documentation for information regarding the additional command-line parameters it supports. After sharing the site, Expose will display the sharable URL that you may use on your other devices or amongst team members:

```
cd ~/Sites/laravel

valet expose
```

To stop sharing your site, you may press `Control + C`.

Sharing Sites On Your Local Network

Valet restricts incoming traffic to the internal `127.0.0.1` interface by default so that your development machine isn't exposed to security risks from the Internet.

If you wish to allow other devices on your local network to access the Valet sites on your machine via your machine's IP address (eg: `192.168.1.10/application.test`), you will need to manually edit the appropriate Nginx configuration file for that site to remove the restriction on the `listen` directive. You should remove the `127.0.0.1:` prefix on the `listen` directive for ports 80 and 443.

If you have not run `valet secure` on the project, you can open up network access for all non-HTTPS sites by editing the `/usr/local/etc/nginx/valet/valet.conf` file. However, if you're serving the project site over HTTPS (you have run `valet secure` for the site) then you should edit the `~/.config/valet/Nginx/app-name.test` file.

Once you have updated your Nginx configuration, run the `valet restart` command to apply the configuration changes.

Site Specific Environment Variables

Some applications using other frameworks may depend on server environment variables but do not provide a way for those variables to be configured within your project. Valet allows you to configure site specific environment variables by adding a `.valet-env.php` file within the root of your project. This file should return an array of site / environment variable pairs which will be added to the global `$_SERVER` array for each site specified in the array:

```
<?php

return [
    // Set $_SERVER['key'] to "value" for the laravel.test site...
    'laravel' => [
        'key' => 'value',
    ],

    // Set $_SERVER['key'] to "value" for all sites...
    '*' => [
        'key' => 'value',
```

```
    ],
];
```

Proxying Services

Sometimes you may wish to proxy a Valet domain to another service on your local machine. For example, you may occasionally need to run Valet while also running a separate site in Docker; however, Valet and Docker can't both bind to port 80 at the same time.

To solve this, you may use the `proxy` command to generate a proxy. For example, you may proxy all traffic from `http://elasticsearch.test` to `http://127.0.0.1:9200`:

```
valet proxy elasticsearch http://127.0.0.1:9200
```

You may remove a proxy using the `unproxy` command:

```
valet unproxy elasticsearch
```

You may use the `proxies` command to list all site configurations that are proxied:

```
valet proxies
```

Custom Valet Drivers

You can write your own Valet "driver" to serve PHP applications running on a framework or CMS that is not natively supported by Valet. When you install Valet, a `~/.config/valet/Drivers` directory is created which contains a `SampleValetDriver.php` file. This file contains a sample driver implementation to demonstrate how to write a custom driver. Writing a driver only requires you to implement three methods: `serves`, `isStaticFile`, and `frontControllerPath`.

All three methods receive the `$sitePath`, `$siteName`, and `$uri` values as their arguments. The `$sitePath` is the fully qualified path to the site being served on your machine, such as `/Users/Lisa/Sites/my-project`. The `$siteName` is the "host" / "site name" portion of the domain (`my-project`). The `$uri` is the incoming request URI (`/foo/bar`).

Once you have completed your custom Valet driver, place it in the `~/.config/valet/Drivers` directory using the `FrameworkValetDriver.php` naming convention. For example, if you are writing a custom valet driver for WordPress, your filename should be `WordPressValetDriver.php`.

Let's take a look at a sample implementation of each method your custom Valet driver should implement.

The `serves` Method

The `serves` method should return `true` if your driver should handle the incoming request. Otherwise, the method should return `false`. So, within this method you should attempt to determine if the given `$sitePath` contains a project of the type you are trying to serve.

For example, let's imagine we are writing a `WordPressValetDriver`. Our `serves` method might look something like this:

```
/**
 * Determine if the driver serves the request.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return bool
 */
public function serves($sitePath, $siteName, $uri)
{
    return is_dir($sitePath.'/wp-admin');
}
```

The `isStaticFile` Method

The `isStaticFile` should determine if the incoming request is for a file that is "static", such as an image or a stylesheet. If the file is static, the method should return the fully qualified path to the static file on disk. If the incoming request is not for a static file, the method should return `false`:

```
/**
 * Determine if the incoming request is for a static file.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return string|false
 */
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath.'/public/'.$uri)) {
        return $staticFilePath;
    }

    return false;
}
```

> NOTE The `isStaticFile` method will only be called if the `serves` method returns `true` for the incoming request and the request URI is not `/`.

The `frontControllerPath` Method

The `frontControllerPath` method should return the fully qualified path to your application's "front controller", which is typically an "index.php" file or equivalent:

```
/**
 * Get the fully resolved path to the application's front controller.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath.'/public/index.php';
}
```

Local Drivers

If you would like to define a custom Valet driver for a single application, create a `LocalValetDriver.php` file in the application's root directory. Your custom driver may extend the base `ValetDriver` class or extend an existing application specific driver such as the `LaravelValetDriver`:

```
class LocalValetDriver extends LaravelValetDriver
{
    /**
     * Determine if the driver serves the request.
     *
     * @param  string  $sitePath
     * @param  string  $siteName
     * @param  string  $uri
     * @return bool
     */
    public function serves($sitePath, $siteName, $uri)
    {
        return true;
    }

    /**
     * Get the fully resolved path to the application's front controller.
     *
     * @param  string  $sitePath
     * @param  string  $siteName
     * @param  string  $uri
     * @return string
     */
    public function frontControllerPath($sitePath, $siteName, $uri)
    {
        return $sitePath.'/public_html/index.php';
```

```
    }
}
```

Other Valet Commands

Command	Description
`valet forget`	Run this command from a "parked" directory to remove it from the parked directory list.
`valet log`	View a list of logs which are written by Valet's services.
`valet paths`	View all of your "parked" paths.
`valet restart`	Restart the Valet daemon.
`valet start`	Start the Valet daemon.
`valet stop`	Stop the Valet daemon.
`valet trust`	Add sudoers files for Brew and Valet to allow Valet commands to be run without prompting for your password.
`valet uninstall`	Uninstall Valet: shows instructions for manual uninstall. Pass the `--force` option to aggressively delete all of Valet's resources.

Valet Directories & Files

You may find the following directory and file information helpful while troubleshooting issues with your Valet environment:

`~/.config/valet`

Contains all of Valet's configuration. You may wish to maintain a backup of this directory.

`~/.config/valet/dnsmasq.d/`

This directory contains DNSMasq's configuration.

`~/.config/valet/Drivers/`

This directory contains Valet's drivers. Drivers determine how a particular framework / CMS is served.

`~/.config/valet/Extensions/`

This directory contains custom Valet extensions / commands.

`~/.config/valet/Nginx/`

This directory contains all of Valet's Nginx site configurations. These files are rebuilt when running the `install`, `secure`, and `tld` commands.

`~/.config/valet/Sites/`

This directory contains all of the symbolic links for your linked projects.

`~/.config/valet/config.json`

This file is Valet's master configuration file.

`~/.config/valet/valet.sock`

This file is the PHP-FPM socket used by Valet's Nginx installation. This will only exist if PHP is running properly.

`~/.config/valet/Log/fpm-php.www.log`

This file is the user log for PHP errors.

`~/.config/valet/Log/nginx-error.log`

This file is the user log for Nginx errors.

`/usr/local/var/log/php-fpm.log`

This file is the system log for PHP-FPM errors.

`/usr/local/var/log/nginx`

This directory contains the Nginx access and error logs.

`/usr/local/etc/php/X.X/conf.d`

This directory contains the `*.ini` files for various PHP configuration settings.

`/usr/local/etc/php/X.X/php-fpm.d/valet-fpm.conf`

This file is the PHP-FPM pool configuration file.

`~/.composer/vendor/laravel/valet/cli/stubs/secure.valet.conf`

This file is the default Nginx configuration used for building SSL certificates for your sites.