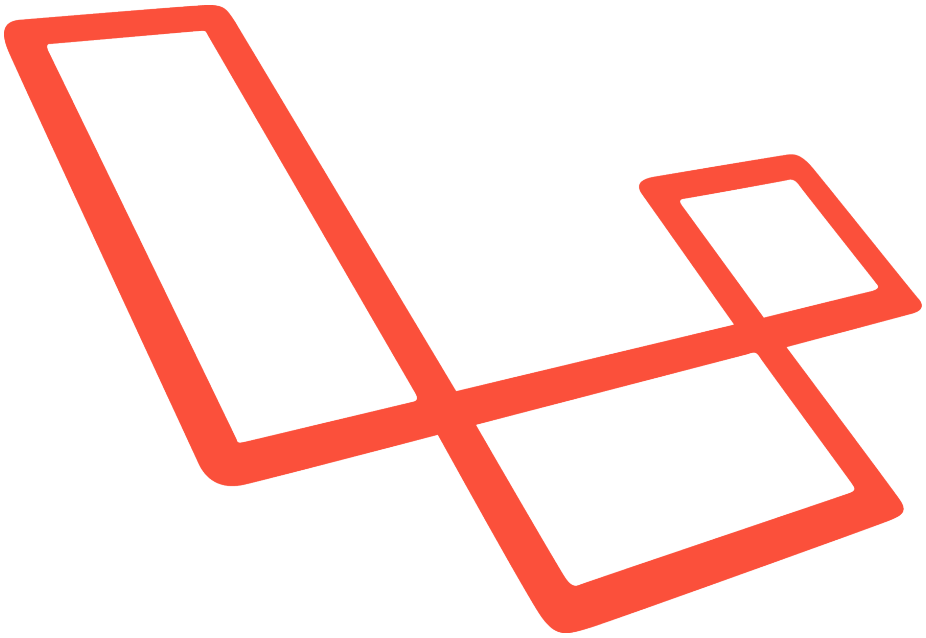
LARAVEL 5.4



THE PHP FRAMEWORK FOR WEB ARTISANS.

JUST MILLION TIMES BETTER.

Release Notes

Versioning Scheme

Laravel's versioning scheme maintains the following convention: `paradigm.minor.patch`. Minor framework releases are released every six months (January and July), while patch releases may be released as often as every week. Patch releases should **never** contain breaking changes.

When referencing the Laravel framework or it's components from your application or package, you should always use a version constraint such as `5.4.*`, since minor releases of Laravel do include breaking changes. However, we strive to always ensure you may update to a new minor release in one day or less.

Paradigm shifting releases are separated by many years and represent fundamental shifts in the framework's architecture and conventions. Currently, there is no paradigm shifting release under development.

Why Doesn't Laravel Use Semantic Versioning?

On one hand, all optional components of Laravel (Cashier, Dusk, Valet, Socialite, etc.) **do** use semantic versioning. However, the Laravel framework itself does not. The reason for this is because semantic versioning is a "reductionist" way of determining if two pieces of code are compatible. Even when using semantic versioning, you still must install the upgraded package and run your automated test suite to know if anything is *actually* incompatible with your code base.

So, instead, the Laravel framework uses a versioning scheme that is more communicative of the actual scope of the release. Furthermore, since patch releases **never** contain intentional breaking changes, you should never receive a breaking change as long as your version constraints follow the `paradigm.minor.*` convention.

Support Policy

For LTS releases, such as Laravel 5.1, 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 6 months and security fixes are provided for 1 year.

Laravel 5.4

Laravel 5.4 continues the improvements made in Laravel 5.3 by adding support for Markdown based emails and notifications, the Laravel Dusk browser automation and testing framework, Laravel Mix, Blade "components" and "slots", route model binding on broadcast channels, higher order messages for Collections, object-based Eloquent events, job-level "retry" and "timeout" settings, "realtime" facades, improved support for Redis Cluster, custom pivot table models, middleware for request input trimming and cleaning, and more. In addition, the entire codebase of the framework was reviewed and refactored for general cleanliness.

> {tip} This documentation summarizes the most notable improvements to the framework; however, more thorough change logs are always available on GitHub.

Markdown Mail & Notifications

> {video} There is a free video tutorial for this feature available on Laracasts.

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. For example, a Markdown email might look something like the following:

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Next Steps:
```

```
- Track Your Order On Our Website
- Pre-Sign For Delivery

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

Using this simple Markdown template, Laravel is able to generate a responsive HTML email and plain-text counterpart:

To read more about Markdown mail and notifications, check out the full mail and notification documentation.

> {tip} 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.

Laravel Dusk

> {video} There is a free video tutorial for this feature available on Laracasts.

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 machine. Instead, Dusk uses a standalone ChromeDriver installation. However, you are free to utilize any other Selenium compatible driver you wish.

Since Dusk operates using a real browser, you are able to easily test and interact with your applications that heavily use JavaScript:

```
/**
 * A basic browser test example.
 *
 * @return void
 */
public function testBasicExample()
{
    $user = factory(User::class)->create([
        'email' => 'taylor@laravel.com',
    ]);

    $this->browse(function ($browser) use ($user) {
        $browser->loginAs($user)
                ->visit('/home')
                ->press('Create Playlist')
                ->whenAvailable('.playlist-modal', function ($modal) {
                    $modal->type('name', 'My Playlist')
```

```
                        ->press('Create');
                });

            $browser->waitForText('Playlist Created');
        });
}
```

For more information on Dusk, consult the full Dusk documentation.

Laravel Mix

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel Mix is the spiritual successor of Laravel Elixir, and its entirely based on Webpack instead of Gulp. Laravel Mix provides a fluent API for defining Webpack build steps for your Laravel application using several common CSS and JavaScript pre-processors. Through simple method chaining, you can fluently define your asset pipeline. For example:

```
mix.js('resources/assets/js/app.js', 'public/js')
   .sass('resources/assets/sass/app.scss', 'public/css');
```

Blade Components & Slots

> {video} There is a free video tutorial for this feature available on Laracasts.

Blade components and slots provide similar benefits to sections and layouts; however, some may find the mental model of components and slots easier to understand. First, let's imagine a reusable "alert" component we would like to reuse throughout our application:

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

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

The `{{ $slot }}` variable will contain the content we wish to inject into the component. Now, to construct this component, we can use the `@component` Blade directive:

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Named slots allow you to provide multiple slots into a single component:

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

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

Named slots may be injected using the `@slot` directive. Any content is not within a `@slot` directive will be passed to the component in the `$slot` variable:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

To read more about components and slots, consult the full Blade documentation.

Broadcast Model Binding

Just like HTTP routes, channel routes may now take advantage of implicit and explicit route model binding. For example, instead of receiving the string or numeric order ID, you may request an actual `Order` model instance:

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

To read more about broadcast model binding, consult the full event broadcasting documentation.

Collection Higher Order Messages

> {video} There is a free video tutorial for this feature available on Laracasts.

Collections now provide support for "higher order messages", which are shortcuts for performing common actions on collections. The collection methods that provide higher order messages are: `contains`, `each`, `every`, `filter`, `first`, `map`, `partition`, `reject`, `sortBy`, `sortByDesc`, and `sum`.

Each higher order message can be accessed as a dynamic property on a collection instance. For instance, let's use the `each` higher order message to call a method on each object within a collection:

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Likewise, we can use the `sum` higher order message to gather the total number of "votes" for a collection of users:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Object Based Eloquent Events

> {video} There is a free video tutorial for this feature available on Laracasts.

Eloquent event handlers may now be mapped to event objects. This provides a more intuitive way of handling Eloquent events and makes it easier to test the events. To get started, define an `$events` property on your Eloquent model that maps various points of the Eloquent model's lifecycle to your own event classes:

```
<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $events = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
```

```
}
```

Job Level Retry & Timeout

Previously, queue job "retry" and "timeout" settings could only be configured globally for all jobs on the command line. However, in Laravel 5.4, these settings may be configured on a per-job basis by defining them directly on the job class:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;

    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

For more information about these settings, consult the full queue documentation.

Request Sanitization Middleware

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel 5.4 includes two new middleware in the default middleware stack: `TrimStrings` and `ConvertEmptyStringsToNull`:

```
/**
 * The application's global HTTP middleware stack.
 *
 * These middleware are run during every request to your application.
 *
 * @var array
 */
```

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

These middleware will automatically trim request input values and convert any empty strings to `null`. This helps you normalize the input for every request entering into your application and not have to worry about continually calling the `trim` function in every route and controller.

"Realtime" Facades

> {video} There is a free video tutorial for this feature available on Laracasts.

Previously, only Laravel's own built-in services exposed facades, which provide quick, terse access to their methods via the service container. However, in Laravel 5.4, you may easily convert any of your application's classes into a facade in realtime simply by prefixing the imported class name with `Facades`. For example, imagine your application contains a class like the following:

```
<?php

namespace App\Services;

class PaymentGateway
{
    protected $tax;

    /**
     * Create a new payment gateway instance.
     *
     * @param  TaxCalculator  $tax
     * @return void
     */
    public function __construct(TaxCalculator $tax)
    {
        $this->tax = $tax;
    }

    /**
     * Pay the given amount.
     *
     * @param  int  $amount
```

```
     * @return void
     */
    public function pay($amount)
    {
        // Pay an amount...
    }
}
```

You may easily use this class as a facade like so:

```
use Facades\ {
    App\Services\PaymentGateway
};

Route::get('/pay/{amount}', function ($amount) {
    PaymentGateway::pay($amount);
});
```

Of course, if you leverage a realtime facade in this way, you may easily write a test for the interaction using Laravel's facade mocking capabilities:

```
PaymentGateway::shouldReceive('pay')->with('100');
```

Custom Pivot Table Models

In Laravel 5.3, all "pivot" table models for `belongsToMany` relationships used the same built-in `Pivot` model instance. In Laravel 5.4, you may define custom models for your pivot tables. If you would like to define a custom model to represent the intermediate table of your relationship, use the `using` method when defining the relationship:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\UserRole');
    }
}
```

Improved Redis Cluster Support

Previously, it was not possible to define Redis connections to single hosts and to clusters in the same application. In Laravel 5.4, you may now define Redis connections to multiple single hosts and multiple clusters within the same application. For more information on Redis in Laravel, please consult the full Redis documentation.

Migration Default String Length

Laravel 5.4 uses the `utf8mb4` character set by default, which includes support for storing "emojis" in the database. If you are upgrading your application from Laravel 5.3, you are not required to switch to this character set.

If you choose to switch to this character set manually and are running a version of MySQL older than the 5.7.7 release, you may need to manually configure the default string length generated by migrations. You may configure this by calling the `Schema::defaultStringLength` method within your `AppServiceProvider`:

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Laravel 5.3

Laravel 5.3 continues the improvements made in Laravel 5.2 by adding a driver based notification system, robust realtime support via Laravel Echo, painless OAuth2 servers via Laravel Passport, full-text model searching via Laravel Scout, Webpack support in Laravel Elixir, "mailable" objects, explicit separation of `web` and `api` routes, Closure based console commands, convenient helpers for storing uploaded files, support for POPO and single-action controllers, improved default frontend scaffolding, and more.

Notifications

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel Notifications provide a simple, expressive API for sending notifications across a variety of delivery channels such as email, Slack, SMS, and more. For example, you may define a notification that an invoice has been paid and deliver that notification via email and SMS. Then, you may send the notification using a single, simple method:

```
$user->notify(new InvoicePaid($invoice));
```

There is already a wide variety of community written drivers for notifications, including support for iOS and Android notifications. To learn more about notifications, be sure to check out the full notification documentation.

WebSockets / Event Broadcasting

While event broadcasting existed in previous versions of Laravel, the Laravel 5.3 release greatly improves this feature of the framework by adding channel-level authentication for private and presence WebSocket channels:

```
/*
 * Authenticate the channel subscription...
 */
Broadcast::channel('orders.*', function ($user, $orderId) {
    return $user->placedOrder($orderId);
});
```

Laravel Echo, a new JavaScript package installable via NPM, has also been released to provide a simple, beautiful API for subscribing to channels and listening for your server-side events in your client-side JavaScript application. Echo includes support for Pusher and Socket.io:

```
Echo.channel('orders.' + orderId)
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.description);
    });
```

In addition to subscribing to traditional channels, Laravel Echo also makes it a breeze to subscribe to presence channels which provide information about who is listening on a given channel:

```
Echo.join('chat.' + roomId)
    .here((users) => {
        //
    })
    .joining((user) => {
```

```
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    });
```

To learn more about Echo and event broadcasting, check out the full documentation.

Laravel Passport (OAuth2 Server)

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel 5.3 makes API authentication a breeze using Laravel Passport, which 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 Alex Bilbie.

Passport makes it painless to issue access tokens via OAuth2 authorization codes. You may also allow your users to create "personal access tokens" via your web UI. To get you started quickly, Passport includes Vue components that can serve as a starting point for your OAuth2 dashboard, allowing users to create clients, revoke access tokens, and more:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

If you do not want to use the Vue components, you are welcome to provide your own frontend dashboard for managing clients and access tokens. Passport exposes a simple JSON API that you may use with any JavaScript framework you choose.

Of course, Passport also makes it simple to define access token scopes that may be requested by application's consuming your API:

```
Passport::tokensCan([
    'place-orders' => 'Place new orders',
    'check-status' => 'Check order status',
]);
```

In addition, Passport includes helpful middleware for verifying that an access token authenticated request contains the necessary token scopes:

```
Route::get('/orders/{order}/status', function (Order $order) {
    // Access token has "check-status" scope...
})->middleware('scope:check-status');
```

Lastly, Passport includes support for consuming your own API from your JavaScript application without worrying about passing access tokens. Passport achieves this through encrypted JWT cookies and synchronized CSRF tokens, allowing you to focus on what matters: your application. For more information on Passport, be sure to check out its full documentation.

Search (Laravel Scout)

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.

Making models searchable is as simple as adding a `Searchable` trait to the model:

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;
}
```

Once the trait has been added to your model, its information will be kept in sync with your search indexes by simply saving the model:

```
$order = new Order;

// ...

$order->save();
```

Once your models have been indexed, its a breeze to perform full-text searches across all of your models. You may even paginate your search results:

```
return Order::search('Star Trek')->get();

return Order::search('Star Trek')->where('user_id', 1)->paginate();
```

Of course, Scout has many more features which are covered in the full documentation.

Mailable Objects

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel 5.3 ships with support for mailable objects. These objects allow you to represent your email messages as a simple objects instead of customizing mail messages within Closures. For example, you may define a simple mailable object for a "welcome" email:

```
class WelcomeMessage extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.welcome');
    }
}
```

Once the mailable object has been defined, you can send it to a user using a simple, expressive API. Mailable objects are great for discovering the intent of your messages while scanning your code:

```
Mail::to($user)->send(new WelcomeMessage);
```

Of course, you may also mark mailable objects as "queueable" so that they will be sent in the background by your queue workers:

```
class WelcomeMessage extends Mailable implements ShouldQueue
{
    //
}
```

For more information on mailable objects, be sure to check out the mail documentation.

Storing Uploaded Files

> {video} There is a free video tutorial for this feature available on Laracasts.

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as profile pictures, photos, and documents. Laravel 5.3

makes it very easy to store uploaded files using the new `store` method on an uploaded file instance. Simply call the `store` method with the path at which you wish to store the uploaded file:

```
/**
 * Update the avatar for the user.
 *
 * @param  Request  $request
 * @return Response
 */
public function update(Request $request)
{
    $path = $request->file('avatar')->store('avatars', 's3');

    return $path;
}
```

For more information on storing uploaded files, check out the full documentation.

Webpack & Laravel Elixir

Along with Laravel 5.3, Laravel Elixir 6.0 has been released with baked-in support for the Webpack and Rollup JavaScript module bundlers. By default, the Laravel 5.3 `gulpfile.js` file now uses Webpack to compile your JavaScript. The full Laravel Elixir documentation contains more information on both of these bundlers:

```
elixir(mix => {
    mix.sass('app.scss')
       .webpack('app.js');
});
```

Frontend Structure

> {video} There is a free video tutorial for this feature available on Laracasts.

Laravel 5.3 ships with a more modern frontend structure. This primarily affects the `make:auth` authentication scaffolding. Instead of loading frontend assets from a CDN, dependencies are specified in the default `package.json` file.

In addition, support for single file Vue components is now included out of the box. A sample `Example.vue` component is included in the `resources/assets/js/components` directory. In addition, the new `resources/assets/js/app.js` file bootstraps and configures your JavaScript libraries and, if applicable, Vue components.

This structure provides more guidance on how to begin developing modern, robust JavaScript applications, without requiring your application to use any given JavaScript or CSS framework. For more information on getting started with modern Laravel frontend development, check out the new introductory frontend documentation.

Routes Files

By default, fresh Laravel 5.3 applications contain two HTTP route files in a new top-level `routes` directory. The `web` and `api` route files provide more explicit guidance in how to split the routes for your web interface and your API. The routes in the `api` route file are automatically assigned the `api` prefix by the `RouteServiceProvider`.

Closure Console Commands

In addition to being defined as command classes, Artisan commands may now be defined as simple Closures in the `commands` method of your `app/Console/Kernel.php` file. In fresh Laravel 5.3 applications, the `commands` method loads a `routes/console.php` file which allows you to define your Console commands as route-like, Closure based entry points into your application:

```
Artisan::command('build {project}', function ($project) {
    $this->info('Building project...');
});
```

For more information on Closure commands, check out the full Artisan documentation.

The `$loop` Variable

> {video} There is a free video tutorial for this feature available on Laracasts.

When looping within a Blade template, 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
```

For more information, consult the full Blade documentation.

Laravel 5.2

Laravel 5.2 continues the improvements made in Laravel 5.1 by adding multiple authentication driver support, implicit model binding, simplified Eloquent global scopes, opt-in authentication scaffolding, middleware groups, rate limiting middleware, array validation improvements, and more.

Authentication Drivers / "Multi-Auth"

In previous versions of Laravel, only the default, session-based authentication driver was supported out of the box, and you could not have more than one authenticatable model instance per application.

However, in Laravel 5.2, you may define additional authentication drivers as well define multiple authenticatable models or user tables, and control their authentication process separately from each other. For example, if your application has one database table for "admin" users and one database table for "student" users, you may now use the `Auth` methods to authenticate against each of these tables separately.

Authentication Scaffolding

Laravel already makes it easy to handle authentication on the back-end; however, Laravel 5.2 provides a convenient, lightning-fast way to scaffold the authentication views for your front-end. Simply execute the `make:auth` command on your terminal:

```
php artisan make:auth
```

This command will generate plain, Bootstrap compatible views for user login, registration, and password reset. The command will also update your routes file with the appropriate routes.

> {note} This feature is only meant to be used on new applications, not during application upgrades.

Implicit Model Binding

Implicit model binding makes it painless to inject relevant models directly into your routes and controllers. For example, assume you have a route defined like the following:

```
use App\User;

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

In Laravel 5.1, you would typically need to use the `Route::model` method to instruct Laravel to inject the `App\User` instance that matches the `{user}` parameter in your route definition. However, in Laravel 5.2, the framework will **automatically** inject this model based on the URI segment, allowing you to quickly gain access to the model instances you need.

Laravel will automatically inject the model when the route parameter segment (`{user}`) matches the route Closure or controller method's corresponding variable name (`$user`) and the variable is type-hinting an Eloquent model class.

Middleware Groups

Middleware groups allow you to group several route middleware under a single, convenient key, allowing you to assign several middleware to a route at once. For example, this can be useful when building a web UI and an API within the same application. You may group the session and CSRF routes into a `web` group, and perhaps the rate limiter in the `api` group.

In fact, the default Laravel 5.2 application structure takes exactly this approach. For example, in the default `App\Http\Kernel.php` file you will find the following:

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

```
    'api' => [
        'throttle:60,1',
    ],
];
```

Then, the `web` group may be assigned to routes like so:

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

However, keep in mind the `web` middleware group is *already* applied to your routes by default since the `RouteServiceProvider` includes it in the default middleware group.

Rate Limiting

A new rate limiter middleware is now included with the framework, allowing you to easily limit the number of requests that a given IP address can make to a route over a specified number of minutes. For example, to limit a route to 60 requests every minute from a single IP address, you may do the following:

```
Route::get('/api/users', ['middleware' => 'throttle:60,1', function () {
    //
}]);
```

Array Validation

Validating array form input fields is much easier in Laravel 5.2. For example, to validate that each e-mail in a given array input field is unique, you may do the following:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users'
]);
```

Likewise, you may use the `*` character when specifying your validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ]
],
```

Bail Validation Rule

A new `bail` validation rule has been added, which instructs the validator to stop validating after the first validation failure for a given rule. For example, you may now prevent the validator from running a `unique` check if an attribute fails an `integer` check:

```
$this->validate($request, [
    'user_id' => 'bail|integer|unique:users'
]);
```

Eloquent Global Scope Improvements

In previous versions of Laravel, global Eloquent scopes were complicated and error-prone to implement; however, in Laravel 5.2, global query scopes only require you to implement a single, simple method: `apply`.

For more information on writing global scopes, check out the full Eloquent documentation.

Laravel 5.1.11

Laravel 5.1.11 introduces authorization support out of the box! Conveniently organize your application's authorization logic using simple callbacks or policy classes, and authorize actions using simple, expressive methods.

For more information, please refer to the authorization documentation.

Laravel 5.1.4

Laravel 5.1.4 introduces simple login throttling to the framework. Consult the authentication documentation for more information.

Laravel 5.1

Laravel 5.1 continues the improvements made in Laravel 5.0 by adopting PSR-2 and adding event broadcasting, middleware parameters, Artisan improvements, and more.

PHP 5.5.9+

Since PHP 5.4 will enter "end of life" in September and will no longer receive security updates from the PHP development team, Laravel 5.1 requires PHP

5.5.9 or greater. PHP 5.5.9 allows compatibility with the latest versions of popular PHP libraries such as Guzzle and the AWS SDK.

LTS

Laravel 5.1 is the first release of Laravel to receive **long term support**. Laravel 5.1 will receive bug fixes for 2 years and security fixes for 3 years. This support window is the largest ever provided for Laravel and provides stability and peace of mind for larger, enterprise clients and customers.

PSR-2

The PSR-2 coding style guide has been adopted as the default style guide for the Laravel framework. Additionally, all generators have been updated to generate PSR-2 compatible syntax.

Documentation

Every page of the Laravel documentation has been meticulously reviewed and dramatically improved. All code examples have also been reviewed and expanded to provide more relevance and context.

Event Broadcasting

In many modern web applications, web sockets 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.

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

To learn more about event broadcasting, check out the event documentation.

Middleware Parameters

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        return $next($request);
    }

}
```

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

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

For more information on middleware, check out the middleware documentation.

Testing Overhaul

The built-in testing capabilities of Laravel have been dramatically improved. A variety of new methods provide a fluent, expressive interface for interacting with your application and examining its responses. For example, check out the following test:

```
public function testNewUserRegistration()
{
    $this->visit('/register')
```

```
            ->type('Taylor', 'name')
            ->check('terms')
            ->press('Register')
            ->seePageIs('/dashboard');
}
```

For more information on testing, check out the testing documentation.

Model Factories

Laravel now ships with an easy way to create stub Eloquent models using model factories. Model factories allow you to easily define a set of "default" attributes for your Eloquent model, and then generate test model instances for your tests or database seeds. Model factories also take advantage of the powerful Faker PHP library for generating random attribute data:

```
$factory->define(App\User::class, function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'remember_token' => str_random(10),
    ];
});
```

For more information on model factories, check out the documentation.

Artisan Improvements

Artisan commands may now be defined using a simple, route-like "signature", which provides an extremely simple interface for defining command line arguments and options. For example, you may define a simple command and its options like so:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--force}';
```

For more information on defining Artisan commands, consult the Artisan documentation.

Folder Structure

To better express intent, the `app/Commands` directory has been renamed to `app/Jobs`. Additionally, the `app/Handlers` directory has been consolidated into a single `app/Listeners` directory which simply contains event listeners. However, this is not a breaking change and you are not required to update to the new folder structure to use Laravel 5.1.

Encryption

In previous versions of Laravel, encryption was handled by the `mcrypt` PHP extension. However, beginning in Laravel 5.1, encryption is handled by the `openssl` extension, which is more actively maintained.

Laravel 5.0

Laravel 5.0 introduces a fresh application structure to the default Laravel project. This new structure serves as a better foundation for building a robust application in Laravel, as well as embraces new auto-loading standards (PSR-4) throughout the application. First, let's examine some of the major changes:

New Folder Structure

The old `app/models` directory has been entirely removed. Instead, all of your code lives directly within the `app` folder, and, by default, is organized to the `App` namespace. This default namespace can be quickly changed using the new `app:name` Artisan command.

Controllers, middleware, and requests (a new type of class in Laravel 5.0) are now grouped under the `app/Http` directory, as they are all classes related to the HTTP transport layer of your application. Instead of a single, flat file of route filters, all middleware are now broken into their own class files.

A new `app/Providers` directory replaces the `app/start` files from previous versions of Laravel 4.x. These service providers provide various bootstrapping functions to your application, such as error handling, logging, route loading, and more. Of course, you are free to create additional service providers for your application.

Application language files and views have been moved to the `resources` directory.

Contracts

All major Laravel components implement interfaces which are located in the `illuminate/contracts` repository. This repository has no external dependencies. Having a convenient, centrally located set of interfaces you may use for decoupling and dependency injection will serve as an easy alternative option to Laravel Facades.

For more information on contracts, consult the full documentation.

Route Cache

If your application is made up entirely of controller routes, you may utilize the new `route:cache` Artisan command to drastically speed up the registration of your routes. This is primarily useful on applications with 100+ routes and will **drastically** speed up this portion of your application.

Route Middleware

In addition to Laravel 4 style route "filters", Laravel 5 now supports HTTP middleware, and the included authentication and CSRF "filters" have been converted to middleware. Middleware provides a single, consistent interface to replace all types of filters, allowing you to easily inspect, and even reject, requests before they enter your application.

For more information on middleware, check out the documentation.

Controller Method Injection

In addition to the existing constructor injection, you may now type-hint dependencies on controller methods. The service container will automatically inject the dependencies, even if the route contains other parameters:

```
public function createPost(Request $request, PostRepository $posts)
{
    //
}
```

Authentication Scaffolding

User registration, authentication, and password reset controllers are now included out of the box, as well as simple corresponding views, which are located at `resources/views/auth`. In addition, a "users" table migration has been included with the framework. Including these simple resources allows rapid

development of application ideas without bogging down on authentication boilerplate. The authentication views may be accessed on the `auth/login` and `auth/register` routes. The `App\Services\Auth\Registrar` service is responsible for user validation and creation.

Event Objects

You may now define events as objects instead of simply using strings. For example, check out the following event:

```
<?php

class PodcastWasPurchased
{
    public $podcast;

    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }
}
```

The event may be dispatched like normal:

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

Of course, your event handler will receive the event object instead of a list of data:

```
<?php

class ReportPodcastPurchase
{
    public function handle(PodcastWasPurchased $event)
    {
        //
    }
}
```

For more information on working with events, check out the full documentation.

Commands / Queueing

In addition to the queue job format supported in Laravel 4, Laravel 5 allows you to represent your queued jobs as simple command objects. These commands live in the `app/Commands` directory. Here's a sample command:

```php
<?php

class PurchasePodcast extends Command implements SelfHandling, ShouldBeQueued
{
    use SerializesModels;

    protected $user, $podcast;

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

    /**
     * Execute the command.
     *
     * @return void
     */
    public function handle()
    {
        // Handle the logic to purchase the podcast...

        event(new PodcastWasPurchased($this->user, $this->podcast));
    }
}
```

The base Laravel controller utilizes the new `DispatchesCommands` trait, allowing you to easily dispatch your commands for execution:

```
$this->dispatch(new PurchasePodcastCommand($user, $podcast));
```

Of course, you may also use commands for tasks that are executed synchronously (are not queued). In fact, using commands is a great way to encapsulate complex tasks your application needs to perform. For more information, check out the command bus documentation.

Database Queue

A `database` queue driver is now included in Laravel, providing a simple, local queue driver that requires no extra package installation beyond your database

software.

Laravel Scheduler

In the past, developers have generated a Cron entry for each console command they wished to schedule. However, this is a headache. Your console schedule is no longer in source control, and you must SSH into your server to add the Cron entries. Let's make our lives easier. The Laravel command scheduler allows you to fluently and expressively define your command schedule within Laravel itself, and only a single Cron entry is needed on your server.

It looks like this:

```
$schedule->command('artisan:command')->dailyAt('15:00');
```

Of course, check out the full documentation to learn all about the scheduler!

Tinker / Psysh

The `php artisan tinker` command now utilizes Psysh by Justin Hileman, a more robust REPL for PHP. If you liked Boris in Laravel 4, you're going to love Psysh. Even better, it works on Windows! To get started, just try:

```
php artisan tinker
```

DotEnv

Instead of a variety of confusing, nested environment configuration directories, Laravel 5 now utilizes DotEnv by Vance Lucas. This library provides a super simple way to manage your environment configuration, and makes environment detection in Laravel 5 a breeze. For more details, check out the full configuration documentation.

Laravel Elixir

Laravel Elixir, by Jeffrey Way, provides a fluent, expressive interface to compiling and concatenating your assets. If you've ever been intimidated by learning Grunt or Gulp, fear no more. Elixir makes it a cinch to get started using Gulp to compile your Less, Sass, and CoffeeScript. It can even run your tests for you!

For more information on Elixir, check out the full documentation.

Laravel Socialite

Laravel Socialite is an optional, Laravel 5.0+ compatible package that provides totally painless authentication with OAuth providers. Currently, Socialite supports Facebook, Twitter, Google, and GitHub. Here's what it looks like:

```
public function redirectForAuth()
{
    return Socialize::with('twitter')->redirect();
}

public function getUserFromProvider()
{
    $user = Socialize::with('twitter')->user();
}
```

No more spending hours writing OAuth authentication flows. Get started in minutes! The full documentation has all the details.

Flysystem Integration

Laravel now includes the powerful Flysystem filesystem abstraction library, providing pain free integration with local, Amazon S3, and Rackspace cloud storage - all with one, unified and elegant API! Storing a file in Amazon S3 is now as simple as:

```
Storage::put('file.txt', 'contents');
```

For more information on the Laravel Flysystem integration, consult the full documentation.

Form Requests

Laravel 5.0 introduces **form requests**, which extend the `Illuminate\Foundation\Http\FormRequest` class. These request objects can be combined with controller method injection to provide a boiler-plate free method of validating user input. Let's dig in and look at a sample `FormRequest`:

```
<?php

namespace App\Http\Requests;

class RegisterRequest extends FormRequest
{
    public function rules()
    {
        return [
```

```
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:8',
        ];
    }

    public function authorize()
    {
        return true;
    }
}
```

Once the class has been defined, we can type-hint it on our controller action:

```
public function register(RegisterRequest $request)
{
    var_dump($request->input());
}
```

When the Laravel service container identifies that the class it is injecting is a `FormRequest` instance, the request will **automatically be validated**. This means that if your controller action is called, you can safely assume the HTTP request input has been validated according to the rules you specified in your form request class. Even more, if the request is invalid, an HTTP redirect, which you may customize, will automatically be issued, and the error messages will be either flashed to the session or converted to JSON. **Form validation has never been more simple.** For more information on `FormRequest` validation, check out the documentation.

Simple Controller Request Validation

The Laravel 5 base controller now includes a `ValidatesRequests` trait. This trait provides a simple `validate` method to validate incoming requests. If `FormRequests` are a little too much for your application, check this out:

```
public function createPost(Request $request)
{
    $this->validate($request, [
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
}
```

If the validation fails, an exception will be thrown and the proper HTTP response will automatically be sent back to the browser. The validation errors will even be flashed to the session! If the request was an AJAX request, Laravel even takes care of sending a JSON representation of the validation errors back to you.

For more information on this new method, check out the documentation.

New Generators

To complement the new default application structure, new Artisan generator commands have been added to the framework. See `php artisan list` for more details.

Configuration Cache

You may now cache all of your configuration in a single file using the `config:cache` command.

Symfony VarDumper

The popular `dd` helper function, which dumps variable debug information, has been upgraded to use the amazing Symfony VarDumper. This provides color-coded output and even collapsing of arrays. Just try the following in your project:

```
dd([1, 2, 3]);
```

Laravel 4.2

The full change list for this release by running the `php artisan changes` command from a 4.2 installation, or by viewing the change file on Github. These notes only cover the major enhancements and changes for the release.

> {note} During the 4.2 release cycle, many small bug fixes and enhancements were incorporated into the various Laravel 4.1 point releases. So, be sure to check the change list for Laravel 4.1 as well!

PHP 5.4 Requirement

Laravel 4.2 requires PHP 5.4 or greater. This upgraded PHP requirement allows us to use new PHP features such as traits to provide more expressive interfaces for tools like Laravel Cashier. PHP 5.4 also brings significant speed and performance improvements over PHP 5.3.

Laravel Forge

Laravel Forge, a new web based application, provides a simple way to create and manage PHP servers on the cloud of your choice, including Linode, DigitalOcean, Rackspace, and Amazon EC2. Supporting automated Nginx configuration, SSH key access, Cron job automation, server monitoring via NewRelic & Papertrail,

"Push To Deploy", Laravel queue worker configuration, and more, Forge provides the simplest and most affordable way to launch all of your Laravel applications.

The default Laravel 4.2 installation's `app/config/database.php` configuration file is now configured for Forge usage by default, allowing for more convenient deployment of fresh applications onto the platform.

More information about Laravel Forge can be found on the official Forge website.

Laravel Homestead

Laravel Homestead is an official Vagrant environment for developing robust Laravel and PHP applications. The vast majority of the boxes' provisioning needs are handled before the box is packaged for distribution, allowing the box to boot extremely quickly. Homestead includes Nginx 1.6, PHP 5.6, MySQL, Postgres, Redis, Memcached, Beanstalk, Node, Gulp, Grunt, & Bower. Homestead includes a simple `Homestead.yaml` configuration file for managing multiple Laravel applications on a single box.

The default Laravel 4.2 installation now includes an `app/config/local/database.php` configuration file that is configured to use the Homestead database out of the box, making Laravel initial installation and configuration more convenient.

The official documentation has also been updated to include Homestead documentation.

Laravel Cashier

Laravel Cashier is a simple, expressive library for managing subscription billing with Stripe. With the introduction of Laravel 4.2, we are including Cashier documentation along with the main Laravel documentation, though installation of the component itself is still optional. This release of Cashier brings numerous bug fixes, multi-currency support, and compatibility with the latest Stripe API.

Daemon Queue Workers

The Artisan `queue:work` command now supports a `--daemon` option to start a worker in "daemon mode", meaning the worker will continue to process jobs without ever re-booting the framework. This results in a significant reduction in CPU usage at the cost of a slightly more complex application deployment process.

More information about daemon queue workers can be found in the queue documentation.

Mail API Drivers

Laravel 4.2 introduces new Mailgun and Mandrill API drivers for the `Mail` functions. For many applications, this provides a faster and more reliable method of sending e-mails than the SMTP options. The new drivers utilize the Guzzle 4 HTTP library.

Soft Deleting Traits

A much cleaner architecture for "soft deletes" and other "global scopes" has been introduced via PHP 5.4 traits. This new architecture allows for the easier construction of similar global traits, and a cleaner separation of concerns within the framework itself.

More information on the new `SoftDeletingTrait` may be found in the Eloquent documentation.

Convenient Auth & Remindable Traits

The default Laravel 4.2 installation now uses simple traits for including the needed properties for the authentication and password reminder user interfaces. This provides a much cleaner default `User` model file out of the box.

"Simple Paginate"

A new `simplePaginate` method was added to the query and Eloquent builder which allows for more efficient queries when using simple "Next" and "Previous" links in your pagination view.

Migration Confirmation

In production, destructive migration operations will now ask for confirmation. Commands may be forced to run without any prompts using the `--force` command.

Laravel 4.1

Full Change List

The full change list for this release by running the `php artisan changes` command from a 4.1 installation, or by viewing the change file on Github. These notes only cover the major enhancements and changes for the release.

New SSH Component

An entirely new `SSH` component has been introduced with this release. This feature allows you to easily SSH into remote servers and run commands. To learn more, consult the SSH component documentation.

The new `php artisan tail` command utilizes the new SSH component. For more information, consult the `tail` command documentation.

Boris In Tinker

The `php artisan tinker` command now utilizes the Boris REPL if your system supports it. The `readline` and `pcntl` PHP extensions must be installed to use this feature. If you do not have these extensions, the shell from 4.0 will be used.

Eloquent Improvements

A new `hasManyThrough` relationship has been added to Eloquent. To learn how to use it, consult the Eloquent documentation.

A new `whereHas` method has also been introduced to allow retrieving models based on relationship constraints.

Database Read / Write Connections

Automatic handling of separate read / write connections is now available throughout the database layer, including the query builder and Eloquent. For more information, consult the documentation.

Queue Priority

Queue priorities are now supported by passing a comma-delimited list to the `queue:listen` command.

Failed Queue Job Handling

The queue facilities now include automatic handling of failed jobs when using the new `--tries` switch on `queue:listen`. More information on handling failed jobs can be found in the queue documentation.

Cache Tags

Cache "sections" have been superseded by "tags". Cache tags allow you to assign multiple "tags" to a cache item, and flush all items assigned to a single tag. More information on using cache tags may be found in the cache documentation.

Flexible Password Reminders

The password reminder engine has been changed to provide greater developer flexibility when validating passwords, flashing status messages to the session, etc. For more information on using the enhanced password reminder engine, consult the documentation.

Improved Routing Engine

Laravel 4.1 features a totally re-written routing layer. The API is the same; however, registering routes is a full 100% faster compared to 4.0. The entire engine has been greatly simplified, and the dependency on Symfony Routing has been minimized to the compiling of route expressions.

Improved Session Engine

With this release, we're also introducing an entirely new session engine. Similar to the routing improvements, the new session layer is leaner and faster. We are no longer using Symfony's (and therefore PHP's) session handling facilities, and are using a custom solution that is simpler and easier to maintain.

Doctrine DBAL

If you are using the `renameColumn` function in your migrations, you will need to add the `doctrine/dbal` dependency to your `composer.json` file. This package is no longer included in Laravel by default.

Upgrade Guide

Upgrading To 5.4.0 From 5.3

Estimated Upgrade Time: 1-2 Hours

> {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.

Updating Dependencies

Update your `laravel/framework` dependency to `5.4.*` in your `composer.json` file. In addition, you should update your `phpunit/phpunit` dependency to `~5.7`.

Removing Compiled Services File

If it exists, you may delete the `bootstrap/cache/compiled.php` file. It is no longer used by the framework.

Flushing The Cache

After upgrading all packages, you should run `php artisan view:clear` to avoid Blade errors related to the removal of `Illuminate\View\Factory::getFirstLoop()`. In addition, you may need to run `php artisan route:clear` to flush the route cache.

Laravel Cashier

Laravel Cashier is already compatible with Laravel 5.4.

Laravel Passport

Laravel Passport `2.0.0` has been released to provide compatibility with Laravel 5.4 and the Axios JavaScript library. If you are upgrading from Laravel 5.3 and using the pre-built Passport Vue components, you should make sure the Axios library is globally available to your application as `axios`.

Laravel Scout

Laravel Scout `3.0.0` has been released to provide compatibility with Laravel 5.4.

Laravel Socialite

Laravel Socialite `3.0.0` has been released to provide compatibility with Laravel 5.4.

Laravel Tinker

In order to continue using the `tinker` Artisan command, you should also install the `laravel/tinker` package:

```
composer require laravel/tinker
```

Once the package has been installed, you should add `Laravel\Tinker\TinkerServiceProvider::class` to the `providers` array in your `config/app.php` configuration file.

Guzzle

Laravel 5.4 requires Guzzle 6.0 or greater.

Authorization

The `getPolicyFor` Method

Previous, when calling the `Gate::getPolicyFor($class)` method, an exception was thrown if no policy could be found. Now, the method will return `null` if no policy is found for the given class. If you call this method directly, make sure you refactor your code to check for `null`:

“‘php $policy = Gate::getPolicyFor($class);

if ($policy) { // code that was previously in the try block } else { // code that was previously in the catch block } “‘

Blade

`@section` Escaping

In Laravel 5.4, inline content passed to a section is automatically escaped:

```
@section('title', $content)
```

If you would like to render unescaped content in a section, you must declare the section using the traditional “long form” style:

```
@section('title')
    {!! $content !!}
@stop
```

Bootstrappers

If you are manually overriding the `$bootstrappers` array on your HTTP or Console kernel, you should rename the `DetectEnvironment` entry to `LoadEnvironmentVariables` and remove `ConfigureLogging`.

Broadcasting

Channel Model Binding

When defining channel name placeholders in Laravel 5.3, the `*` character is used. In Laravel 5.4, you should define these placeholders using `{foo}` style placeholders, like routes:

```
Broadcast::channel('App.User.{userId}', function ($user, $userId) {
    return (int) $user->id === (int) $userId;
});
```

Collections

The `every` Method

The behavior of the `every` method has been moved to the `nth` method to match the method name defined by Lodash.

The `random` Method

Calling `$collection->random(1)` will now return a new collection instance with one item. Previously, this would return a single object. This method will only return a single object if no arguments are supplied.

Container

Aliasing Via `bind` / `instance`

In previous Laravel releases, you could pass an array as the first parameter to the `bind` or `instance` methods to register an alias:

```
$container->bind(['foo' => FooContract::class], function () {
    return 'foo';
});
```

However, this behavior has been removed in Laravel 5.4. To register an alias, you should now use the `alias` method:

```
$container->alias(FooContract::class, 'foo');
```

Binding Classes With Leading Slashes

Binding classes into the container with leading slashes is no longer supported. This feature required a significant amount of string formatting calls to be made within the container. Instead, simply register your bindings without a leading slash:

```
$container->bind('Class\Name', function () {
    //
});

$container->bind(ClassName::class, function () {
    //
});
```

`make` Method Parameters

The container's `make` method no longer accepts a second array of parameters. This feature typically indicates a code smell. Typically, you can always construct the object in another way that is more intuitive.

Resolving Callbacks

The container's `resolving` and `afterResolving` method now must be provided a class name or binding key as the first argument to the method:

```
$container->resolving('Class\Name', function ($instance) {
    //
});

$container->afterResolving('Class\Name', function ($instance) {
    //
});
```

`share` Method Removed

The `share` method has been removed from the container. This was a legacy method that has not been documented in several years. If you are using this method, you should begin using the `singleton` method instead:

```
$container->singleton('foo', function () {
    return 'foo';
});
```

Console

The `Illuminate\Console\AppNamespaceDetectorTrait` Trait

If you are directly referencing the `Illuminate\Console\AppNamespaceDetectorTrait` trait, update your code to reference `Illuminate\Console\DetectsApplicationNamespace` instead.

Database

Custom Connections

If you were previously binding a service container binding for a `db.connection.{driver-name}` key in order to resolve a custom database connection instance, you should now use the `Illuminate\Database\Connection::resolverFor` method in the `register` method of your `AppServiceProvider`:

```
use Illuminate\Database\Connection;

Connection::resolverFor('driver-name', function ($connection, $database, $prefix, $config) {
    //
});
```

Fetch Mode

Laravel no longer includes the ability to customize the PDO "fetch mode" from your configuration files. Instead, `PDO::FETCH_OBJ` is always used. If you would still like to customize the fetch mode for your application you may listen for the new `Illuminate\Database\Events\StatementPrepared` event:

```
Event::listen(StatementPrepared::class, function ($event) {
    $event->statement->setFetchMode(...);
});
```

Eloquent

Date Casts

The `date` cast now converts the column to a `Carbon` object and calls the `startOfDay` method on the object. If you would like to preserve the time portion of the date, you should use the `datetime` cast.

Foreign Key Conventions

If the foreign key is not explicitly specified when defining a relationship, Eloquent will now use the table name and primary key name for the related model to build the foreign key. For the vast majority of applications, this is not a change of behavior. For example:

```
public function user()
{
    return $this->belongsTo(User::class);
}
```

Just like previous Laravel releases, this relationship will typically use `user_id` as the foreign key. However, the behavior could be different from previous releases if you are overriding the `getKeyName` method of the `User` model. For example:

```
public function getKeyName()
{
    return 'key';
}
```

When this is the case, Laravel will now respect your customization and determine the foreign key column name is `user_key` instead of `user_id`.

Has One / Many `createMany`

The `createMany` method of a `hasOne` or `hasMany` relationship now returns a collection object instead of an array.

Related Model Connections

Related models will now use the same connection as the parent model. For example, if you execute a query like:

```
User::on('example')->with('posts');
```

Eloquent will query the posts table on the `example` connection instead of the default database connection. If you want to read the `posts` relationship from the default connection, you should to explicitly set the model's connection to your application's default connection.

The `create` & `forceCreate` Methods

The `Model::create` & `Model:: forceCreate` methods have been moved to the `Illuminate\Database\Eloquent\Builder` class in order to provide better support for creating models on multiple connections. However, if you are extending these methods in your own models, you will need to modify your implementation to call the `create` method on the builder. For example:

```
public static function create(array $attributes = [])
{
    $model = static::query()->create($attributes);

    // ...

    return $model;
}
```

The `hydrate` Method

If you are currently passing a custom connection name to this method, you should now use the `on` method:

```
User::on('connection')->hydrate($records);
```

`hydrateRaw` Method

The `Model::hydrateRaw` method has been renamed to `fromQuery`. If you are passing a custom connection name to this method, you should now use the `on` method:

```
User::on('connection')->fromQuery('...');
```

The `whereKey` Method

The `whereKey($id)` method will now add a "where" clause for the given primary key value. Previously, this would fall into the dynamic "where" clause builder and add a "where" clause for the "key" column. If you used the `whereKey` method to dynamically add a condition for the `key` column you should now use `where('key', ...)` instead.

The `factory` Helper

Calling `factory(User::class, 1)->make()` or `factory(User::class, 1)->create()` will now return a collection with one item. Previously, this would return a single model. This method will only return a single model if the amount is not supplied.

Events

Contract Changes

If you are manually implementing the `Illuminate\Contracts\Events\Dispatcher` interface in your application or package, you should rename the `fire` method to `dispatch`.

Event Priority

Support for event handler "priorities" has been removed. This undocumented feature typically indicates an abuse of the event feature. Instead, consider using a series of synchronous method calls. Alternatively, you may dispatch a new event from within the handler of another event in order to ensure that a given event's handler fires after an unrelated handler.

Wildcard Event Handler Signatures

Wildcard event handlers now receive the event name as their first argument and the array of event data as their second argument. The `Event::firing` method has been removed:

```
Event::listen('*', function ($eventName, array $data) {
    //
});
```

The `kernel.handled` Event

The `kernel.handled` event is now an object based event using the `Illuminate\Foundation\Http\Events\RequestHandled` class.

The `locale.changed` Event

The `locale.changed` event is now an object based event using the `Illuminate\Foundation\Events\LocaleUpdated` class.

The `illuminate.log` Event

The `illuminate.log` event is now an object based event using the `Illuminate\Log\Events\MessageLogged` class.

Exceptions

The `Illuminate\Http\Exception\HttpResponseException` has been renamed to `Illuminate\Http\Exceptions\HttpResponseException`. Note that `Exceptions` is now plural. Likewise, the `Illuminate\Http\Exception\PostTooLargeException` has been renamed to `Illuminate\Http\Exceptions\PostTooLargeException`.

Mail

`Class@method` Syntax

Sending mail using `Class@method` syntax is no longer supported. For example:

```
Mail::send('view.name', $data, 'Class@send');
```

If you are sending mail in this way you should convert these calls to mailables.

New Configuration Options

In order to provide support for Laravel 5.4's new Markdown mail components, you should add the following block of configuration to the bottom of your `mail` configuration file:

```
'markdown' => [
    'theme' => 'default',

    'paths' => [
        resource_path('views/vendor/mail'),
    ],
],
```

Queueing Mail With Closures

In order to queue mail, you now must use a mailable. Queuing mail using the `Mail::queue` and `Mail::later` methods no longer supports using Closures to configure the mail message. This feature required the use of special libraries to serialize Closures since PHP does not natively support this feature.

Redis

Improved Clustering Support

Laravel 5.4 introduces improved Redis cluster support. If you are using Redis clusters, you should place your cluster connections inside of a `clusters` configuration option in the Redis portion of your `config/database.php` configuration file:

```
'redis' => [

    'client' => 'predis',

    'options' => [
        'cluster' => 'redis',
    ],

    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', '127.0.0.1'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
```

```
    ],

],
```

Routing

Post Size Middleware

The class `Illuminate\Foundation\Http\Middleware\VerifyPostSize` has been renamed to `Illuminate\Foundation\Http\Middleware\ValidatePostSize`.

The `middleware` Method

The `middleware` method of the `Illuminate\Routing\Router` class has been renamed to `aliasMiddleware()`. It is likely that most applications never call this method manually, as it is typically only called by the HTTP kernel to register route-level middleware defined in the `$routeMiddleware` array.

`Route` Methods

The `getUri` method of the `Illuminate\Routing\Route` class has been removed. You should use the `uri` method instead.

The `getMethods` method of the `Illuminate\Routing\Route` class has been removed. You should use the `methods` method instead.

The `getParameter` method of the `Illuminate\Routing\Route` class has been removed. You should use the `parameter` method instead.

The `getPath` method of the `Illuminate\Routing\Route` class has been removed. You should use the `uri` method instead.

Sessions

Symfony Compatibility

Laravel's session handlers no longer implements Symfony's `SessionInterface`. Implementing this interface required us to implement extraneous features that were not needed by the framework. Instead, a new `Illuminate\Contracts\Session\Session` interface has been defined and may be used instead. The following code changes should also be applied:

All calls to the `->set()` method should be changed to `->put()`. Typically, Laravel applications would never call the `set` method since it has never been documented within the Laravel documentation. However, it is included here out of caution.

All calls to the `->getToken()` method should be changed to `->token()`.

All calls to the `$request->setSession()` method should be changed to `setLaravelSession()`.

Testing

Laravel 5.4's testing layer has been re-written to be simpler and lighter out of the box. If you would like to continue using the testing layer present in Laravel 5.3, you may install the `laravel/browser-kit-testing` package into your application. This package provides full compatibility with the Laravel 5.3 testing layer. In fact, you can run the Laravel 5.4 testing layer side-by-side with the Laravel 5.3 testing layer.

In order to allow Laravel to autoload any new tests you generate using the Laravel 5.4 test generators, you should add the `Tests` namespace to your `composer.json` file's `autoload-dev` block:

```
"psr-4": {
    "Tests\\": "tests/"
}
```

Running Laravel 5.3 & 5.4 Tests In A Single Application

First install the `laravel/browser-kit-testing` package:

```
composer require laravel/browser-kit-testing --dev
```

Once the package has been installed, create a copy of your `tests/TestCase.php` file and save it to your `tests` directory as `BrowserKitTestCase.php`. Then, modify the file to extend the `Laravel\BrowserKitTesting\TestCase` class. Once you have done this, you should have two base test classes in your `tests` directory: `TestCase.php` and `BrowserKitTestCase.php`. In order for your `BrowserKitTestCase` class to be properly loaded, you may need to add it to your `composer.json` file:

```
"autoload-dev": {
    "classmap": [
        "tests/TestCase.php",
        "tests/BrowserKitTestCase.php"
    ]
},
```

Tests written on Laravel 5.3 will extend the `BrowserKitTestCase` class while any new tests that use the Laravel 5.4 testing layer will extend the `TestCase` class. Your `BrowserKitTestCase` class should look like the following:

```
<?php
```

```
use Illuminate\Contracts\Console\Kernel;
use Laravel\BrowserKitTesting\TestCase as BaseTestCase;

abstract class BrowserKitTestCase extends BaseTestCase
{
    /**
     * The base URL of the application.
     *
     * @var string
     */
    public $baseUrl = 'http://localhost';

    /**
     * Creates the application.
     *
     * @return \Illuminate\Foundation\Application
     */
    public function createApplication()
    {
        $app = require __DIR__.'/../bootstrap/app.php';

        $app->make(Kernel::class)->bootstrap();

        return $app;
    }
}
```

Once you have created this class, make sure to update all of your tests to extend your new `BrowserKitTestCase` class. This will allow all of your tests written on Laravel 5.3 to continue running on Laravel 5.4. If you choose, you can slowly begin to port them over to the new Laravel 5.4 test syntax or Laravel Dusk.

> {note} If you are writing new tests and want them to use the Laravel 5.4 testing layer, make sure to extend the `TestCase` class.

Installing Dusk In An Upgraded Application

If you would like to install Laravel Dusk into an application that has been upgraded from Laravel 5.3, first install it via Composer:

```
composer require laravel/dusk
```

Next, you will need to create a `CreatesApplication` trait in your `tests` directory. This trait is responsible for creating fresh application instances for test cases. The trait should look like the following:

```
<?php
```

```
use Illuminate\Contracts\Console\Kernel;

trait CreatesApplication
{
    /**
     * Creates the application.
     *
     * @return \Illuminate\Foundation\Application
     */
    public function createApplication()
    {
        $app = require __DIR__.'/../bootstrap/app.php';

        $app->make(Kernel::class)->bootstrap();

        return $app;
    }
}
```

> {note} If you have namespaced your tests and are using the PSR-4 autoloading standard to load your `tests` directory, you should place the `CreatesApplication` trait under the appropriate namespace.

Once you have completed these preparatory steps, you can follow the normal Dusk installation instructions.

Environment

The Laravel 5.4 test class no longer manually forces `putenv('APP_ENV=testing')` for each test. Instead, the framework utilizes the `APP_ENV` variable from the loaded `.env` file.

Event Fake

The `Event` fake's `assertFired` method should be updated to `assertDispatched`, and the `assertNotFired` method should be updated to `assertNotDispatched`. The method's signatures have not been changed.

Mail Fake

The `Mail` fake has been greatly simplified for the Laravel 5.4 release. Instead of using the `assertSentTo` method, you should now simply use the `assertSent` method and utilize the `hasTo`, `hasCc`, etc. helper methods within your callback:

```
Mail::assertSent(MailableName::class, function ($mailable) {
    return $mailable->hasTo('email@example.com');
});
```

Translation

`{Inf}` Placeholder

If you are using the `{Inf}` placeholder for pluralizing your translation strings, you should update your translation strings to use the `*` character instead:

```
{0} First Message|{1,*} Second Message
```

URL Generation

The `forceSchema` Method

The `forceSchema` method of the `Illuminate\Routing\UrlGenerator` class has been renamed to `forceScheme`.

Validation

Date Format Validation

Date format validation is now more strict and supports the placeholders present within the documentation for the PHP date function. In previous releases of Laravel, the timezone placeholder `P` would accept all timezone formats; however, in Laravel 5.4 each timezone format has a unique placeholder as per the PHP documentation.

Method Names

The `addError` method has been renamed to `addFailure`. In addition, the `doReplacements` method has been renamed to `makeReplacements`. Typically, these changes will only be relevant if you are extending the `Validator` class.

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.

Contribution Guide

- Bug Reports

- Core Development Discussion
- Which Branch?
- Security Vulnerabilities
- Coding Style
  - PHPDoc
  - StyleCI

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.

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 Cashier
- Laravel Cashier for Braintree
- Laravel Envoy
- Laravel Framework
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Passport
- Laravel Scout
- Laravel Socialite
- Laravel Website

Core Development Discussion

You may propose new features or improvements of existing Laravel behavior in the Laravel Internals 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 LaraChat Slack team. 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 (5.1). 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 backwards compatible** with the current Laravel release may be sent to the latest stable branch.

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

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

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  $shared
 * @return void
 */
public function bind($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.

Installation

Installation

> {video} Are you a visual learner? Laracasts provides a free, thorough introduction to Laravel for newcomers to the framework. It's a great place to start your journey.

Server Requirements

The Laravel framework has a few system requirements. Of course, all of these requirements are satisfied by the Laravel Homestead virtual machine, so it's highly recommended that you use Homestead as your local Laravel development environment.

However, if you are not using Homestead, you will need to make sure your server meets the following requirements:

- PHP >= 5.6.4
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension

- Tokenizer PHP Extension
- XML PHP Extension

Installing Laravel

Laravel utilizes Composer to manage its dependencies. So, before using Laravel, make sure you have Composer installed on your machine.

Via Laravel Installer

First, download the Laravel installer using Composer:

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

Make sure to place the `$HOME/.composer/vendor/bin` directory (or the equivalent directory for your OS) in your $PATH so the `laravel` executable can be located by your system.

Once installed, the `laravel new` command will create a fresh Laravel installation in the directory you specify. For instance, `laravel new blog` will create a directory named `blog` containing a fresh Laravel installation with all of Laravel's dependencies already installed:

```
laravel new blog
```

Via Composer Create-Project

Alternatively, you may also install Laravel by issuing the Composer `create-project` command in your terminal:

```
composer create-project --prefer-dist laravel/laravel blog
```

Local Development Server

If you have PHP installed locally and you would like to use PHP's built-in development server to serve your application, you may use the `serve` Artisan command. This command will start a development server at `http://localhost:8000`:

```
php artisan serve
```

Of course, more robust local development options are available via Homestead and Valet.

Configuration

Public Directory

After installing Laravel, you should configure your web server's document / web root to be the `public` directory. The `index.php` in this directory serves as the front controller for all HTTP requests entering your application.

Configuration Files

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.

Directory Permissions

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

Application Key

The next thing you should do after installing Laravel is set your application key to a random string. If you installed Laravel via Composer or the Laravel installer, this key has already been set for you by the `php artisan key:generate` command.

Typically, this string should be 32 characters long. The key can be set in the `.env` environment file. If you have not renamed the `.env.example` file to `.env`, you should do that now. **If the application key is not set, your user sessions and other encrypted data will not be secure!**

Additional Configuration

Laravel needs almost no other 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.

You may also want to configure a few additional components of Laravel, such as:

- Cache
- Database
- Session

Web Server Configuration

Pretty URLs

Apache

Laravel includes a `public/.htaccess` file that is used to provide URLs without the `index.php` front controller in the path. Before serving Laravel with Apache, be sure to enable the `mod_rewrite` module so the `.htaccess` file will be honored by the server.

If the `.htaccess` file that ships with Laravel does not work with your Apache installation, try this alternative:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

If you are using Nginx, the following directive in your site configuration will direct all requests to the `index.php` front controller:

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

Of course, when using Homestead or Valet, pretty URLs will be automatically configured.

Configuration

Introduction

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

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 by Vance Lucas. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file. If you install Laravel via Composer, this file will automatically be renamed to `.env`. Otherwise, you should rename the file manually.

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 gain access to your source control repository, since any sensitive credentials would get exposed.

If you are developing with a team, you may wish to continue including a `.env.example` file with your application. By putting place-holder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application. You may also create a `.env.testing` file. This file will override values from the `.env` file when running PHPUnit tests or executing Artisan commands with the `--env=testing` option.

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

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 several of the options 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 used 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:

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

You may also pass arguments to the `environment` method to check 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 overriden by a server-level `APP_ENV` environment variable. This can be useful when you need to share the same application for different environment configurations, so you can set up a given host to match a given environment in your server's configurations.

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');
```

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 will be loaded quickly by the framework.

You should typically run the `php artisan config:cache` command as part of your production deployment routine. 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.

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, simply execute the `down` Artisan command:

```
php artisan down
```

You may also provide `message` and `retry` options to the `down` command. The `message` value may be used to display or log a custom message, while the `retry` value will be set as the `Retry-After` HTTP header's value:

```
php artisan down --message="Upgrading Database" --retry=60
```

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

```
php artisan up
```

Maintenance Mode Response Template

The default template for maintenance mode responses is located in `resources/views/errors/503.blade.php`. You are free to modify this view as needed for your application.

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 Envoyer to accomplish zero-downtime deployment with Laravel.

Directory Structure

Introduction

The default Laravel application structure is intended to provide a great starting point for both large and small applications. Of course, 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.

Where Is The Models Directory?

When getting started with Laravel, many developers are confused by the lack of a `models` directory. However, the lack of such a directory is intentional. We find the word "models" ambiguous since it means many different things to many different people. Some developers refer to an application's "model" as the totality

of all of its business logic, while others refer to "models" as classes that interact with a relational database.

For this reason, we choose to place Eloquent models in the `app` directory by default, and allow the developer to place them somewhere else if they choose.

The Root Directory

The App Directory

The `app` directory, as you might expect, 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 files that bootstrap the framework and configure autoloading. This directory also houses a `cache` directory which contains framework generated files for performance optimization such as the route and services cache files.

The Config Directory

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

The Database Directory

The `database` directory contains your database migration 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. 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 LESS, SASS, 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, 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, which provides rate limiting. 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 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` command.

The Tests Directory

The `tests` directory contains your automated tests. An example PHPUnit is 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.

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 simply 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 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, as you might expect, 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 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. For more information, check out the authorization documentation.

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.

Laravel Homestead

Introduction

Laravel strives to make the entire PHP development experience delightful, including your local development environment. Vagrant provides a simple, elegant way to manage and provision Virtual Machines.

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. No more worrying about messing up your operating system! Vagrant boxes are completely disposable. If something goes wrong, you can destroy and re-create the box in minutes!

Homestead runs on any Windows, Mac, or Linux system, and includes the Nginx web server, PHP 7.1, MySQL, Postgres, Redis, Memcached, Node, and all of the other goodies 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 16.04
- Git
- PHP 7.1
- Nginx
- MySQL
- MariaDB
- Sqlite3
- Postgres
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- ngrok

Installation & Setup

First Steps

Before launching your Homestead environment, you must install VirtualBox 5.1, VMWare, or Parallels as well as Vagrant. 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.

Installing The Homestead Vagrant Box

Once VirtualBox / VMware and Vagrant have been installed, you should add the `laravel/homestead` box to your Vagrant installation using 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.

Installing Homestead

You may install Homestead by simply cloning the repository. Consider cloning the repository into a `Homestead` folder within your "home" directory, as the Homestead box will serve as the host to all of your Laravel projects:

```
cd ~

git clone https://github.com/laravel/homestead.git Homestead
```

You should check out a tagged version of Homestead since the `master` branch may not always be stable. You can find the latest stable version on the Github Release Page:

```
cd Homestead

// Clone the desired release...
git checkout v4.0.5
```

Once you have cloned the Homestead repository, run the `bash init.sh` command from the Homestead directory to create the `Homestead.yaml` configuration file. The `Homestead.yaml` file will be placed in the Homestead directory:

```
// Mac / 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`, or `parallels`. You may set this to the provider you prefer:

```
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 environment. You may configure as many shared folders as necessary:

```
folders:
    - map: ~/Code
      to: /home/vagrant/Code
```

To enable NFS, just add a simple flag to your synced folder configuration:

```
folders:
    - map: ~/Code
      to: /home/vagrant/Code
      type: "nfs"
```

You may also pass any options supported by Vagrant's Synced Folders by listing them under the `options` key:

```
folders:
    - map: ~/Code
      to: /home/vagrant/Code
      type: "rsync"
      options:
          rsync__args: ["--verbose", "--archive", "--delete", "-zz"]
          rsync__exclude: ["node_modules"]
```

Configuring Nginx Sites

Not familiar with Nginx? No problem. The `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 project you are working on:

```
sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public
```

If you change the `sites` property after provisioning the Homestead box, you should re-run `vagrant reload --provision` to update the Nginx configuration on the virtual machine.

The Hosts File

You must add the "domains" for your Nginx sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead sites into your Homestead machine. On Mac 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.app
```

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.app
```

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 box 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 simply `vagrant up`.

To install Homestead directly into your project, require it using Composer:

```
composer require laravel/homestead --dev
```

Once Homestead has been installed, use the `make` command to generate the `Vagrantfile` and `Homestead.yaml` file in your project root. The `make` command will automatically configure the `sites` and `folders` directives in the `Homestead.yaml` file.

Mac / 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.app` in your browser. Remember, you will still need to add an `/etc/hosts` file entry for `homestead.app` or the domain of your choice.

Installing MariaDB

If you prefer to use MariaDB instead of MySQL, you may add the `mariadb` option to your `Homestead.yaml` file. This option will remove MySQL and install MariaDB. MariaDB serves as a drop-in replacement for MySQL so you should still use the `mysql` database driver in your application's database configuration:

```
box: laravel/homestead
ip: "192.168.20.20"
memory: 2048
cpus: 4
provider: virtualbox
mariadb: true
```

Daily Usage

Accessing Homestead Globally

Sometimes you may want to `vagrant up` your Homestead machine from anywhere on your filesystem. You can do this on Mac / Linux systems by adding a Bash function to your Bash profile. On Windows, you may accomplish this by adding a "batch" file to your `PATH`. These scripts will allow you to run any Vagrant command from anywhere on your system and will automatically point that command to your Homestead installation:

Mac / Linux

```
function homestead() {
    ( cd ~/Homestead && vagrant $* )
}
```

Make sure to tweak the `~/Homestead` path in the function to the location of your actual Homestead installation. Once the function is installed, you may run commands like `homestead up` or `homestead ssh` from anywhere on your system.

Windows

Create a `homestead.bat` batch file anywhere on your machine with the following contents:

```
@echo off

set cwd=%cd%
set homesteadVagrant=C:\Homestead

cd /d %homesteadVagrant% && vagrant %*
cd /d %cwd%

set cwd=
set homesteadVagrant=
```

Make sure to tweak the example `C:\Homestead` path in the script to the actual location of your Homestead installation. After creating the file, add the file location to your `PATH`. You may then run commands like `homestead up` or `homestead ssh` from anywhere on your system.

Connecting Via SSH

You can SSH into your virtual machine by issuing the `vagrant ssh` terminal command from your Homestead directory.

But, since you will probably need to SSH into your Homestead machine frequently, consider adding the "function" described above to your host machine to quickly SSH into the Homestead box.

Connecting To Databases

A `homestead` database is configured for both MySQL and Postgres out of the box. For even more convenience, Laravel's `.env` file configures the framework to use this database out of the box.

To connect to your MySQL or Postgres database from your host machine's database client, you should connect to `127.0.0.1` and port `33060` (MySQL) or `54320` (Postgres). 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 database configuration file since Laravel is running *within* the virtual machine.

Adding Additional Sites

Once your Homestead environment is provisioned and running, you may want to add additional Nginx sites for your Laravel applications. You can run as many Laravel installations as you wish on a single Homestead environment. To add an additional site, simply add the site to your `Homestead.yaml` file:

```
sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public
    - map: another.app
      to: /home/vagrant/Code/another/public
```

If Vagrant is not automatically managing your "hosts" file, you may need to add the new site to that file as well:

```
192.168.10.10  homestead.app
192.168.10.10  another.app
```

Once the site has been added, run the `vagrant reload --provision` 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 Symfony application to Homestead using the `symfony2` site type:

```
sites:
    - map: symfony2.app
      to: /home/vagrant/Code/Symfony/public
      type: symfony2
```

The available site types are: `apache`, `laravel` (the default), `proxy`, `silverstripe`, `statamic`, and `symfony2`.

Configuring Cron Schedules

Laravel provides a convenient way to schedule Cron jobs by scheduling a single `schedule:run` Artisan command to be run every minute. The `schedule:run` command will examine the job schedule defined in your `App\Console\Kernel` class to determine which jobs should be 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.app
      to: /home/vagrant/Code/Laravel/public
```

```
    schedule: true
```

The Cron job for the site will be defined in the `/etc/cron.d` folder of the virtual machine.

Ports

By default, the following ports are forwarded to your Homestead environment:

- **SSH:** 2222 → Forwards To 22
- **HTTP:** 8000 → Forwards To 80
- **HTTPS:** 44300 → Forwards To 443
- **MySQL:** 33060 → Forwards To 3306
- **Postgres:** 54320 → Forwards To 5432
- **Mailhog:** 8025 → Forwards To 8025

Forwarding Additional Ports

If you wish, you may forward additional ports to the Vagrant box, as well as specify their protocol:

```
ports:
    - send: 93000
      to: 9300
    - send: 7777
      to: 777
      protocol: udp
```

Sharing Your Environment

Sometimes you may wish to share what you're currently working on with coworkers or a client. Vagrant has a built-in way to support this via `vagrant share`; 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 machine via `vagrant ssh` and run `share homestead.app`. This will share the `homestead.app` site from your `Homestead.yaml` configuration file. Of course, you may substitute any of your other configured sites for `homestead.app`:

```
share homestead.app
```

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.app -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.

Network Interfaces

The `networks` property of the `Homestead.yaml` configures network interfaces for your Homestead environment. 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 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)"
```

Updating Homestead

You can update Homestead in two simple steps. First, you should update the Vagrant box using the `vagrant box update` command:

```
vagrant box update
```

Next, you need to update the Homestead source code. If you cloned the repository you can simply `git pull origin master` at the location you originally cloned the repository.

If you have installed Homestead via your project's `composer.json` file, you should ensure your `composer.json` file contains `"laravel/homestead": "^4"` and update your dependencies:

```
composer update
```

Old Versions

You can easily override the version of the box that Homestead uses by adding the following line to your `Homestead.yaml` file:

```
version: 0.6.0
```

An example:

```
box: laravel/homestead
version: 0.6.0
ip: "192.168.20.20"
memory: 2048
cpus: 4
provider: virtualbox
```

When you use an older version of the Homestead box you need to match that with a compatible version of the Homestead source code. Below is a chart which shows the supported box versions, which version of Homestead source code to use, and the version of PHP provided:

	Homestead Version	Box Version
PHP 7.0	3.1.0	0.6.0
PHP 7.1	4.0.0	1.0.0

Provider Specific Settings

VirtualBox

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 lines to your `Homestead.yaml` file:

```
provider: virtualbox
natdnshostresolver: off
```

Laravel Valet

- Introduction
  - Valet Or Homestead
- Installation
  - Upgrading
- Serving Sites

Introduction

Valet is a Laravel development environment for Mac minimalists. No Vagrant, no `/etc/hosts` file. You can even share your sites publicly using local tunnels. *Yeah, we like it too.*

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 `*.dev` domain to point to sites installed on your local machine.

In other words, a blazing fast Laravel development environment that uses roughly 7 MB of RAM. Valet isn't a complete replacement for Vagrant 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
- Symfony
- Zend
- CakePHP 3
- WordPress
- Bedrock
- Craft
- Statamic
- Jigsaw
- Static HTML

However, you may extend Valet with your own custom drivers.

Valet Or Homestead

As you may know, Laravel offers Homestead, another local Laravel development environment. Homestead and Valet differ in regards to their intended audience and their approach to local development. Homestead offers an entire Ubuntu virtual machine with automated Nginx configuration. Homestead is a wonderful choice if you want a fully virtualized Linux development environment or are on Windows / Linux.

Valet only supports Mac, and requires you to install PHP and a database server directly onto your local machine. This is easily achieved by using Homebrew with commands like `brew install php71` and `brew install mysql`. Valet provides a blazing fast local development environment with minimal resource consumption, so it's great for developers who only require PHP / MySQL and do not need a fully virtualized development environment.

Both Valet and Homestead are great choices for configuring your Laravel development environment. Which one you choose will depend on your personal taste and your team's needs.

Installation

Valet requires macOS and Homebrew. Before installation, you should make sure that no other programs such as Apache or Nginx are binding to your local machine's port 80.

- Install or update Homebrew to the latest version using `brew update`.
- Install PHP 7.1 using Homebrew via `brew install homebrew/php/php71`.
- Install Valet with Composer via `composer global require laravel/valet`. Make sure the `~/.composer/vendor/bin` directory is in your system's "PATH".
- Run the `valet install` command. This will configure and install Valet and DnsMasq, and register Valet's daemon to launch when your system starts.

Once Valet is installed, try pinging any `*.dev` domain on your terminal using a command such as `ping foobar.dev`. 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.

Using Another Domain

By default, Valet serves your projects using the `.dev` TLD. If you'd like to use another domain, you can do so using the `valet domain tld-name` command.

For example, if you'd like to use `.app` instead of `.dev`, run `valet domain app` and Valet will start serving your projects at `*.app` automatically.

Database

If you need a database, try MySQL by running `brew install mysql` on your command line. Once MySQL has been installed, you may start it using the `brew`

`services start mysql` command. You can then connect to the database at `127.0.0.1` using the `root` username and an empty string for the password.

Upgrading

You may update your Valet installation using 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.

Upgrading To Valet 2.0

Valet 2.0 transitions Valet's underlying web server from Caddy to Nginx. Before upgrading to this version you should run the following commands to stop and uninstall the existing Caddy daemon:

```
valet stop
valet uninstall
```

Next, you should upgrade to the latest version of Valet. Depending on how you installed Valet, this is typically done through Git or Composer. If you installed Valet via Composer, you should use the following command to update to the latest major version:

```
composer global require laravel/valet
```

Once the fresh Valet source code has been downloaded, you should run the `install` command:

```
valet install
valet restart
```

After upgrading, it may be necessary to re-park or re-link your sites.

Serving Sites

Once Valet is installed, you're ready to start serving sites. Valet provides two commands to help you serve your Laravel sites: `park` and `link`.

The `park` Command {#valet-the-park-command}

- Create a new directory on your Mac by running something like `mkdir ~/Sites`. Next, `cd ~/Sites` and run `valet park`. This command will register your current working directory as a path that Valet should search for sites.
- Next, create a new Laravel site within this directory: `laravel new blog`.
- Open `http://blog.dev` in your browser.

That's all there is to it. Now, any Laravel project you create within your "parked" directory will automatically be served using the `http://folder-name.dev` convention.

The `link` Command {#valet-the-link-command}

The `link` command may also be used to serve your Laravel sites. This command is useful if you want to serve a single site in a directory and not the entire directory.

- To use the command, navigate to one of your projects and run `valet link app-name` in your terminal. Valet will create a symbolic link in `~/.valet/Sites` which points to your current working directory.
- After running the `link` command, you can access the site in your browser at `http://app-name.dev`.

To see a listing of all of your linked directories, run the `valet links` command. You may use `valet unlink app-name` to destroy the symbolic link.

> {tip} You can use `valet link` to serve the same project from multiple (sub)domains. To add a subdomain or another domain to your project run `valet link subdomain.app-name` from the project folder.

Securing Sites With TLS {#valet-securing-sites}

By default, Valet serves sites over plain HTTP. However, if you would like to serve a site over encrypted TLS using HTTP/2, use the `secure` command. For example, if your site is being served by Valet on the `laravel.dev` 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 host name that you wish to unsecure:

```
valet unsecure laravel
```

Sharing Sites

Valet even includes a command to share your local sites with the world. No additional software installation is required once Valet is installed.

To share a site, navigate to the site's directory in your terminal and run the `valet share` command. A publicly accessible URL will be inserted into your clipboard and is ready to paste directly into your browser. That's it.

To stop sharing your site, hit `Control + C` to cancel the process.

> {note} `valet share` does not currently support sharing sites that have been secured using the `valet secure` command.

Custom Valet Drivers

You can write your own Valet "driver" to serve PHP applications running on another framework or CMS that is not natively supported by Valet. When you install Valet, a `~/.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 `~/.valet/Drivers` directory using the `FrameworkValetDriver.php` naming convention. For example, if you are writing a custom valet driver for WordPress, your file name 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 pretend we are writing a `WordPressValetDriver`. Our serve 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 your "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';
}
```

Other Valet Commands

Command	Description
`valet forget`	Run this command from a "parked" directory to remove it from the parked directory list.
`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 uninstall`	Uninstall the Valet daemon entirely.

Request Lifecycle

- Introduction
- Lifecycle Overview
- 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 Things

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 simply 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` script. 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.

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.

The method signature for the HTTP kernel's `handle` method is quite simple: receive a `Request` and return 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. First, the `register` method will be called on all providers, then, once all providers have been registered, the `boot` method will be called.

Service providers are responsible for bootstrapping all of the framework's various components, such as the database, queue, validation, and routing components.

Since they bootstrap and configure every feature offered by the framework, service providers are the most important aspect of the entire Laravel bootstrap process.

Dispatch Request

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.

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. Of course, 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. Of course, for large applications, you may wish to create several service providers, each with a more granular type of bootstrapping.

Service Container

Introduction

The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase

that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Let's look at a simple example:

```
<?php

namespace App\Http\Controllers;

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

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.

Binding

Binding Basics

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

> {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.

Simple Bindings

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:

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

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.

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:

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

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:

```
$api = new HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\Api', $api);
```

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);
```

Binding Interfaces To Implementations

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

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

This 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 a constructor, or any other location where dependencies are injected by the service container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
 * @param  EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
```

```
    $this->pusher = $pusher;
}
```

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, 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 Illuminate\Support\Facades\Storage;
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;

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

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

Tagging

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

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

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

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

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

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

Resolving

The `make` Method

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

```
$api = $this->app->make('HelpSpot\API');
```

If you are in a location of your code that does not have access to the `$app` variable, you may use the global `resolve` helper:

```
$api = resolve('HelpSpot\API');
```

Automatic Injection

Alternatively, and importantly, you may simply "type-hint" the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, queue jobs, middleware, and more. In practice, this is how most of your objects 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\Users\Repository as UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
```

```
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

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

Container Events

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

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

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

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.

Service 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. Of course, many of these 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.

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 Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @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 `Riak\Connection` in the service container. If you don't understand how the service container works, check out its documentation.

The Boot Method

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class 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;

public function boot(ResponseFactory $response)
{
    $response->macro('caps', function ($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, simply 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, set the `defer` property to `true` and define a `provides` method. The `provides` method should return the service container bindings registered by the provider:

```
<?php

namespace App\Providers;

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

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = true;

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

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

```
}
```

Facades

Introduction

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.

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.

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.

> {tip} When building a third-party package that interacts with Laravel, it's better to inject Laravel contracts instead of using facades. Since packages are built outside of Laravel itself, you will not have access to Laravel's facade testing helpers.

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');
});
```

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');

    $this->visit('/cache')
         ->see('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 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');

    $this->visit('/cache')
         ->see('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 method `get` 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.

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`
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	`cache`
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	
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	`hash`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	`auth.password`
Queue	Illuminate\Queue\QueueManager	`queue`
Queue (Instance)	Illuminate\Contracts\Queue\Queue	`queue`
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\Database	`redis`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Contracts\Routing\ResponseFactory	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	

Facade	Class	Service Container Binding
Storage	Illuminate\Contracts\Filesystem\Factory	`filesystem`
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	

Contracts

Introduction

Laravel's Contracts are a set of interfaces that define the core services provided by the framework. For example, a `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 by package developers.

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.

> {tip} Most applications will be fine regardless of whether you prefer facades or contracts. However, if you are building a package, you should strongly consider using contracts since they will be easier to test in a package context.

When To Use Contracts

As discussed elsewhere, much of 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. As long as you are keeping your class' responsibilities focused, you will notice very few practical differences between using contracts and facades.

However, you may still have several questions regarding contracts. For example, why use interfaces at all? Isn't using interfaces more complicated? Let's distill the reasons for using interfaces to the following headings: loose coupling and simplicity.

Loose Coupling

First, let's review some code that is tightly coupled to a cache implementation. Consider the following:

```
<?php

namespace App\Orders;

class Repository
{
    /**
     * The cache instance.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param  \SomePackage\Cache\Memcached  $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
```

```
        $this->cache = $cache;
    }

    /**
     * Retrieve an Order by ID.
     *
     * @param  int  $id
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id))    {
            //
        }
    }
}
```

In this class, the code is tightly coupled to a given cache implementation. It is tightly coupled because we are depending on a concrete Cache class from a package vendor. If the API of that package changes our code must change as well.

Likewise, if we want to replace our underlying cache technology (Memcached) with another technology (Redis), we again will have to modify our repository. Our repository should not have so much knowledge regarding who is providing them data or how they are providing it.

Instead of this approach, we can improve our code by depending on a simple, vendor agnostic interface:

```
<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository
{
    /**
     * The cache instance.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param  Cache  $cache
     * @return void
```

```
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}
```

Now the code is not coupled to any specific vendor, or even Laravel. Since the contracts package contains no implementation and no dependencies, you may easily write an alternative implementation of any given contract, allowing you to replace your cache implementation without modifying any of your cache consuming code.

Simplicity

When all of Laravel's services are neatly defined within simple interfaces, it is very easy to determine the functionality offered by a given service. **The contracts serve as succinct documentation to the framework's features.**

In addition, when you depend on simple interfaces, your code is easier to understand and maintain. Rather than tracking down which methods are available to you within a large, complicated class, you can refer to a simple, clean interface.

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\User;
use App\Events\OrderWasPlaced;
use Illuminate\Contracts\Redis\Database;

class CacheOrderInformation
{
    /**
```

```
     * The Redis database implementation.
     */
    protected $redis;

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

    /**
     * Handle the event.
     *
     * @param  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\Factory	Auth
Illuminate\Contracts\Auth\PasswordBroker	Password
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Broadcasting\Broadcaster	
Illuminate\Contracts\Cache\Repository	Cache
Illuminate\Contracts\Cache\Factory	Cache::driver()
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Container\Container	App

Contract	References Facade
Illuminate\Contracts\Cookie\Factory	Cookie
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Encryption\Encrypter	Crypt
Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	
Illuminate\Contracts\Filesystem\Factory	File
Illuminate\Contracts\Filesystem\Filesystem	File
Illuminate\Contracts\Foundation\Application	App
Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Logging\Log	Log
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Queue\Factory	Queue::driver()
Illuminate\Contracts\Queue\Queue	Queue
Illuminate\Contracts\Redis\Database	Redis
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Validation\Factory	Validator::make()
Illuminate\Contracts\Validation\Validator	
Illuminate\Contracts\View\Factory	View::make()
Illuminate\Contracts\View\View	

Routing

- Form Method Spoofing
- Accessing The Current Route

Basic Routing

The most basic Laravel routes simply accept a URI and a `Closure`, providing a very simple and expressive method of defining routes:

```
Route::get('foo', 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 the framework. 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.

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('foo', function () {
    //
});
```

CSRF Protection

Any HTML forms pointing to `POST`, `PUT`, 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_field() }}
    ...
</form>
```

Route Parameters

Required Parameters

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

```
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. Route parameters may not contain a `-` character. Use an underscore (`_`) instead.

Optional Parameters

Occasionally you may need to specify a route parameter, but make the presence of that route parameter optional. 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]+']);
```

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 `RouteServiceProvider`:

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

    parent::boot();
}
```

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...
});
```

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', 'UserController@showProfile')->name('profile');
```

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via the global `route` function:

```
// 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 URL in their correct positions:

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

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

Route Groups

Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route. Shared attributes are specified in an array format as the first parameter to the `Route::group` method.

Middleware

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

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('/', function ()    {
        // Uses Auth Middleware
```

```
    });

    Route::get('user/profile', function () {
        // Uses Auth Middleware
    });
});
```

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers using the `namespace` parameter in the group array:

```
Route::group(['namespace' => 'Admin'], function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

Remember, by default, the `RouteServiceProvider` includes your route files within a namespace group, allowing you to register controller routes without specifying the full `App\Http\Controllers` namespace prefix. So, you only need to specify the portion of the namespace that comes after the base `App\Http\Controllers` namespace.

Sub-Domain Routing

Route groups may also be used to handle sub-domain routing. Sub-domains may be assigned route parameters just like route URIs, allowing you to capture a portion of the sub-domain for usage in your route or controller. The sub-domain may be specified using the `domain` key on the group attribute array:

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

Route Prefixes

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

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

Route Model Binding

When injecting a model ID to a route or controller action, you will often query 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:

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

Since the `$user` variable is type-hinted as the `App\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.

Customizing The Key Name

If you would like model binding to 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';
}
```

Explicit Binding

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 in the `boot` method of the `RouteServiceProvider` class:

```
public function boot()
{
    parent::boot();

    Route::model('user', App\User::class);
}
```

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

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

Since we have bound all `{user}` parameters to the `App\User` model, a `User` instance will be injected into the route. So, for example, a request to `profile/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 use your own 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:

```
public function boot()
{
    parent::boot();

    Route::bind('user', function ($value) {
        return App\User::where('name', $value)->first();
    });
}
```

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

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

You may use the `method_field` helper to generate the `_method` input:

```
{{ method_field('PUT') }}
```

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:

```
$route = Route::current();

$name = Route::currentRouteName();

$action = Route::currentRouteAction();
```

Refer to the API documentation for both the underlying class of the Route facade and Route instance to review all accessible methods.

Middleware

Introduction

Middleware provide a convenient mechanism for 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 the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Of course, additional middleware can be written to perform a variety of tasks besides authentication. A CORS middleware might be responsible for adding the proper headers to all responses leaving your application. A logging middleware might log all incoming requests to your application.

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 CheckAge
```

This command will place a new `CheckAge` class within your `app/Http/Middleware` directory. In this middleware, we will only allow access to the route if the supplied `age` is greater than 200. Otherwise, we will redirect the users back to the `home` URI.

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->age <= 200) {
            return redirect('home');
        }

        return $next($request);
    }

}
```

As you can see, if the given `age` is less than or equal to `200`, the middleware will return an HTTP redirect to the client; otherwise, the request will be passed further into the application. To pass the request deeper into the application (allowing the middleware to "pass"), simply call the `$next` callback with the `$request`.

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

Before & After Middleware

Whether a middleware runs before or after a request depends on the middleware itself. For example, the following middleware would perform some task **before** the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        // Perform action

        return $response;
    }
}
```

Registering Middleware

Global Middleware

If you want a middleware to run during every HTTP request to your application, simply 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 `app/Http/Kernel.php` file. By default, the `$routeMiddleware` property of this class contains entries for the middleware included with Laravel. To add your own, simply append it to this list and assign it a key of your choosing. For example:

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

protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::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('admin/profile', function () {
    //
})->middleware('auth');
```

You may also assign multiple middleware to the route:

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

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

```
use App\Http\Middleware\CheckAge;

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

Middleware Groups

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

Out of the box, Laravel comes with `web` and `api` middleware groups that contains common middleware you may want to apply to your web UI and API routes:

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

    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

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

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

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

> {tip} Out of the box, the `web` middleware group is automatically applied to your `routes/web.php` file by the `RouteServiceProvider`.

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 `CheckRole` 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 CheckRole
{
    /**
     * 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. For example, the "session" middleware included with Laravel writes the session data to storage after the response has been sent to the browser. If you define a `terminate` method on your middleware, it will automatically be called after the response is sent to the browser.

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Store the session data...
    }
}
```

The `terminate` method should receive both the request and the response. Once you have defined a terminable middleware, you should add it to the list of 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.

CSRF Protection

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

Introduction

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

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 one actually making the requests to the application.

Anytime you define a 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_field` helper to generate the token field:

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

The `VerifyCsrfToken` middleware, which is included in the `web` middleware group, will automatically verify that the token in the request input matches the token stored in the session.

CSRF Tokens & Vue

If you are using the Vue JavaScript framework without the authentication scaffolding provided by the `make:auth` Artisan command, you will need to manually define a `Laravel` JavaScript object in your primary application layout. This object specifies the CSRF token Vue should use when making requests:

```
<script>
    window.Laravel = {!! json_encode([
        'csrfToken' => csrf_token(),
    ]) !!};
</script>
```

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 `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 BaseVerifier;

class VerifyCsrfToken extends BaseVerifier
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
    ];
}
```

X-CSRF-TOKEN

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

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

Then, once you have created the meta tag, 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:

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

X-XSRF-TOKEN

Laravel stores the current CSRF token in a 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 convenience since some JavaScript frameworks, like Angular, automatically place its value in the `X-XSRF-TOKEN` header.

Controllers

Introduction

Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behavior using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the `app/Http/Controllers` directory.

Basic Controllers

Defining Controllers

Below is an example of a basic controller class. Note that the controller extends the base controller class included with Laravel. The base class provides a few convenience methods such as the `middleware` method, which may be used to attach middleware to controller actions:

```
<?php

namespace App\Http\Controllers;

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

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

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

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

Now, when a request matches the specified route URI, the `show` method on the `UserController` class will be executed. Of course, the route parameters will also be passed to the method.

> {tip} Controllers are not **required** to extend a base class. However, you will not have access to convenience features such as the `middleware`, `validate`, and `dispatch` methods.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. Since the `RouteServiceProvider` loads your route files within a route group that contains the namespace, we only specified the portion of the class name that comes after the `App\Http\Controllers` portion of the namespace.

If you choose to nest your controllers deeper into the `App\Http\Controllers` directory, simply use the specific class name relative to the `App\Http\Controllers` root namespace. So, if your full controller class is `App\Http\Controllers\Photos\AdminController`, you should register routes to the controller like so:

```
Route::get('foo', 'Photos\AdminController@method');
```

Single Action Controllers

If you would like to define a controller that only handles a single action, you may place a single `__invoke` method on the controller:

```
<?php
```

```
namespace App\Http\Controllers;

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

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

When registering routes for single action controllers, you do not need to specify a method:

```
Route::get('user/{id}', 'ShowProfile');
```

Controller Middleware

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

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

However, it is more convenient to specify middleware within your controller's constructor. Using the `middleware` method from your controller's constructor, you may easily assign middleware to the controller's action. You may even restrict the middleware to only certain methods on the controller class:

```
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 a middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
    // ...

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

> {tip} You may assign middleware to a subset of controller actions; however, it may indicate your controller is growing too large. Instead, consider breaking your controller into multiple, smaller controllers.

Resource Controllers

Laravel resource routing assigns the typical "CRUD" routes to a controller with a single line of code. For example, you may wish to create a controller that handles all HTTP requests for "photos" stored by your application. Using the `make:controller` Artisan command, we can quickly create such a controller:

```
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 resourceful route to the controller:

```
Route::resource('photos', 'PhotoController');
```

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, including notes informing you of the HTTP verbs and URIs they handle.

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

Verb	URI	Action	Route Name
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
```

Spoofing Form Methods

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_field` helper can create this field for you:

```
{{ method_field('PUT') }}
```

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:

```
Route::resource('photo', 'PhotoController', ['only' => [
    'index', 'show'
]]);

Route::resource('photo', 'PhotoController', ['except' => [
    'create', 'store', 'update', '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 options:

```
Route::resource('photo', 'PhotoController', ['names' => [
    'create' => 'photo.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 by passing `parameters` in the options array. The `parameters` array should be an associative array of resource names and parameter names:

```
Route::resource('user', 'AdminUserController', ['parameters' => [
    'user' => 'admin_user'
]]);
```

The example above generates the following URIs for the resource's `show` route:

```
/user/{admin_user}
```

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 in the `boot` method of your `AppServiceProvider`:

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

/**
 * Bootstrap any application services.
 *
 * @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')` 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 `Route::resource`; otherwise, the routes defined by the `resource` method may unintentionally take precedence over your supplemental routes:

```
Route::get('photos/popular', 'PhotoController@method');

Route::resource('photos', 'PhotoController');
```

> {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  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
```

```
}
```

Of course, you may also type-hint any Laravel contract. If the container can resolve it, you can type-hint it. Depending on your application, injecting your dependencies into your controller may provide better testability.

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, simply list your route arguments after your other dependencies. For example, if your route is defined like so:

```
Route::put('user/{id}', 'UserController@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  Request  $request
     * @param  string  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Route Caching

> {note} Closure based routes cannot be cached. To use route caching, you must convert any Closure routes to controller classes.

If your application is exclusively using controller based routes, 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. In some cases, your route registration may even be up to 100x faster. To generate a route cache, just 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
```

HTTP Requests

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` class on your controller method. The incoming request instance will automatically be injected by the service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

        //
    }
}
```

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:

```
Route::put('user/{id}', 'UserController@update');
```

You may still type-hint the `Illuminate\Http\Request` and access your route parameter `id` by defining your controller method as follows:

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param  Request  $request
     * @param  string  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Accessing The Request Via Route Closures

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) {
    //
});
```

Request Path & Method

The `Illuminate\Http\Request` instance provides a variety of methods for examining the HTTP request for your application 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://domain.com/foo/bar`, the `path` method will return `foo/bar`:

```
$uri = $request->path();
```

The `is` method allows you to verify that the incoming request path matches a given pattern. You may use the `*` character as a wildcard when utilizing this method:

```
if ($request->is('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:

```
// Without Query String...
$url = $request->url();

// With Query String...
$url = $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')) {
    //
}
```

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 zendframework/zend-diactoros
```

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 Trimming & Normalization

By default, Laravel includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the stack by the `App\Http\Kernel` class. 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.

Retrieving Input

Retrieving All Input Data

You may also retrieve all of the input data as an `array` using the `all` method:

```
$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');
```

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 route parameters.

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 dig into JSON arrays:

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

Retrieving A Portion Of The Input Data

If you need to retrieve a subset of the input data, you may use the `only` and `except` methods. Both of these methods accept a single `array` or a dynamic list of arguments:

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

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

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

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

The `only` method returns all of the key / value pairs that you request, even if the key is not present on the incoming request. When the key is not present on the request, the value will be `null`. If you would like to retrieve a portion of input data that is actually present on the request, you may use the `intersect` method:

```
$input = $request->intersect(['username', 'password']);
```

Determining If An Input Value Is Present

You should use the `has` method to determine if a value is present on the request. The `has` method returns `true` if the value is present and is not an empty string:

```
if ($request->has('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 unlikely you will need to manually use these methods, 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('form')->withInput(
    $request->except('password')
);
```

Retrieving Old Input

To retrieve flashed input from the previous request, use the `old` method on the `Request` instance. 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. 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 a `Illuminate\Http\Request` instance:

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

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
);
```

Generating Cookie Instances

If you would like to generate a `Symfony\Component\HttpFoundation\Cookie` instance that can be given 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);
```

Files

Retrieving Uploaded Files

You may access uploaded files from a `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 which will move an uploaded file to one of your disks, which may be a location on your local filesystem or even 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 file name, since a unique ID will automatically be generated to serve as the file name.

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 file name to be automatically generated, you may use the `storeAs` method, which accepts the path, file name, and disk name as its arguments:

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

HTTP Responses

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 simply 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');
});
```

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',
            ]);
```

Attaching Cookies To Responses

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

```
return response($content)
                ->header('Content-Type', $type)
                ->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:

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

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 or has all of the session middleware applied:

```
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 simply 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 should 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. Remember, you do not need to specify the full namespace to the controller since Laravel's `RouteServiceProvider` will automatically set the base controller namespace:

```
return redirect()->action('HomeController@index');
```

If your controller route requires parameters, you may pass them as the second argument to the `action` method:

```
return redirect()->action(
    'UserController@profile', ['id' => 1]
);
```

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 () {
    // Update the user's profile...

    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
```

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 should 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

file name as the second argument to the method, which will determine the file name that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

> {note} Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII file name.

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. For example, from a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * Register the application's response macros.
     *
     * @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. The macro's Closure will be executed when calling the macro name from a `ResponseFactory` implementation or the `response` helper:

```
return response()->caps('foo');
```

Views

- Creating Views
- Passing Data To Views
  - Sharing Data With All Views
- View Composers

Creating Views

> {tip} Looking for more information on how to write Blade templates? Check out the full Blade documentation to get started.

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the `resources/views` directory. A simple view might look something like this:

```
<!-- View stored in resources/views/greeting.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']);
});
```

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.

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

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

Determining If A View Exists

If you need to determine if a view exists, you may use the `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:

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

When passing information in this manner, `$data` should be an array with key/value pairs. Inside your view, you can then access each value using its corresponding key, such as `<?php echo $key; ?>`. As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view:

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

Sharing Data With All Views

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

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
```

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

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

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.

For this example, let's register the view composers within a service provider. We'll use the `View` facade to access the underlying `Illuminate\Contracts\View\Factory` contract implementation. Remember, Laravel does not include a default directory for view composers. You are free to organize them however you wish. For example, you could create an `app/Http/ViewComposers` directory:

```
<?php

namespace App\Providers;

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

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
```

```
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer(
            'profile', 'App\Http\ViewComposers\ProfileComposer'
        );

        // Using Closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }

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

> {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 `ProfileComposer@compose` method will be executed each time the `profile` view is being rendered. So, let's define the composer class:

```
<?php

namespace App\Http\ViewComposers;

use Illuminate\View\View;
use App\Repositories\UserRepository;

class ProfileComposer
{
    /**
     * The user repository implementation.
     *
```

```
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

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

Just before the view is rendered, the composer's `compose` method is called with the `Illuminate\View\View` instance. You may use the `with` method to bind data to the view.

> {tip} 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:

```
View::composer(
    ['profile', 'dashboard'],
    'App\Http\ViewComposers\MyViewComposer'
);
```

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:

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

HTTP Session

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple 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 out of the box.

Configuration

The 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. In

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

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.
- `array` - sessions are stored in a PHP array and will not be persisted.

  {tip} The array driver is 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 items. Below is an example `Schema` declaration for the table:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

You may use the `session:table` Artisan command to generate this migration:

```
php artisan session:table

php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer. You may configure your Redis connections in the `database` configuration file. In the `session` configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

Using 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 controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

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

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

        //
    }
}
```

When you retrieve a value from the session, you may also pass a default value as the second argument to the `get` method. This default value will be returned if the specified key does not exist in the session. If you pass a `Closure` as the default value to the `get` method 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 a value is present in the session, you may use the `has` method. The `has` method returns `true` if the value is present and is not `null`:

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

To determine if a value is present in the session, even if its value is `null`, you may use the `exists` method. The `exists` method returns `true` if the value is present:

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

Storing Data

To store data in the session, you will typically use the `put` method or the `session` helper:

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global 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 only for the next request. You may do so using the `flash` method. Data stored in the session using this method will only be available during the subsequent HTTP request, and then will be deleted. Flash data is primarily useful for short-lived status messages:

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

If you need to keep your flash data around for 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:

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

$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 the built-in `LoginController`; however, if you need to manually regenerate the session ID, you may use the `regenerate` method.

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

Adding Custom Session Drivers

Implementing The Driver

Your custom session driver should implement the `SessionHandlerInterface`. This interface contains just a few simple methods we need to implement. A stubbed MongoDB implementation looks something like this:

```
<?php

namespace App\Extensions;

class MongoHandler 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 `MongoHandler`.

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 leave it as an empty stub. It is

simply a fact of poor interface design (which we'll discuss later) that PHP requires us to implement this method.
- 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 `$sessionId` to some persistent storage system, such as MongoDB, Dynamo, etc. Again, you should not perform any serialization - Laravel will have already handled that for you.
- The `destroy` method should remove the data associated with the `$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 the framework. To add additional drivers to Laravel's session backend, you may use the `extend` method on the `Session` facade. You should call the `extend` method from the `boot` method of a service provider. You may do this from the existing `AppServiceProvider` or create an entirely new provider:

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionStore;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore;
```

```
        });
    }

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

Once the session driver has been registered, you may use the `mongo` driver in your `config/session.php` configuration file.

Validation

Introduction

Laravel provides several different approaches to validate your application's incoming data. By default, Laravel's base controller class uses a `ValidatesRequests` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

Validation Quickstart

To learn about 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.

Defining The Routes

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

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');
```

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

Creating The Controller

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

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
```

```
        return view('post.create');
    }

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

Writing The Validation Logic

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

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

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

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

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

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

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:

```
$this->validate($request, [
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

In this example, if the `required` rule on the `title` attribute fails, the `unique` rule will not be checked. Rules will be validated in the order they are assigned.

A Note On Nested Attributes

If your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request parameters 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 will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our `GET` route. This is because Laravel will check for errors in the session data, and automatically bind them to the view if they are available. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

> {tip} The `$errors` variable is bound to the view 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.

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

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

<h1>Create Post</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

A Note On Optional Fields

By default, Laravel includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the stack by the `App\Http\Kernel` class. Because of this, you will often need to mark your "optional" request fields as `nullable` if you do not want the validator to consider `null` values as invalid. For example:

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

In this example, we are specifying that the `publish_at` field may be either `null` or a valid date representation. If the `nullable` modifier is not added to the rule definition, the validator would consider `null` an invalid date.

Customizing The Flashed Error Format

If you wish to customize the format of the validation errors that are flashed to the session when validation fails, override the `formatValidationErrors` on your base controller. Don't forget to import the `Illuminate\Contracts\Validation\Validator` class at the top of the file:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Contracts\Validation\Validator;
use Illuminate\Routing\Controller as BaseController;
use Illuminate\Foundation\Validation\ValidatesRequests;

abstract class Controller extends BaseController
{
    use DispatchesJobs, ValidatesRequests;

    /**
     * {@inheritdoc}
     */
    protected function formatValidationErrors(Validator $validator)
    {
        return $validator->errors()->all();
    }
}
```

AJAX Requests & Validation

In this example, we used a traditional form to send data to the application. However, many applications use AJAX requests. When using the `validate` method during an AJAX request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

Form Request Validation

Creating Form Requests

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that contain validation logic. To create a form request class, use the `make:request` Artisan CLI command:

```
php artisan make:request StoreBlogPost
```

The generated class will be placed in the `app/Http/Requests` directory. If this directory does not exist, it will be created when you run the `make:request` command. Let's add a few validation rules to the `rules` method:

```
/**
 * Get the validation rules that apply to the request.
```

```
     *
     * @return array
     */
    public function rules()
    {
        return [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ];
    }
```

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method is called, meaning you do not need to clutter your controller with any validation logic:

```
    /**
     * Store the incoming blog post.
     *
     * @param  StoreBlogPost  $request
     * @return Response
     */
    public function store(StoreBlogPost $request)
    {
        // The incoming request is valid...
    }
```

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an AJAX request, a HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

Adding After Hooks To Form Requests

If you would like to add an "after" hook to a form request, you may use the `withValidator` method. This method receives the fully constructed validator, allowing you to call any of its methods before the validation rules are actually evaluated:

```
    /**
     * Configure the validator instance.
     *
     * @param  \Illuminate\Validation\Validator  $validator
     * @return void
     */
    public function withValidator($validator)
```

```
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field!');
        }
    });
}
```

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may check if the authenticated user actually has the authority to update a given resource. For example, you may determine if a user actually owns a blog comment they are attempting to update:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

Since all form requests extend the base Laravel request class, we may use the `user` method to access the currently authenticated user. Also note the call to the `route` method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the `{comment}` parameter in the example below:

```
Route::post('comment/{comment}');
```

If the `authorize` method returns `false`, a HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to have authorization logic in another part of your application, simply return `true` from the `authorize` method:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
```

```
    return true;
}
```

Customizing The Error Format

If you wish to customize the format of the validation errors that are flashed to the session when validation fails, override the `formatErrors` on your base request (`App\Http\Requests\Request`). Don't forget to import the `Illuminate\Contracts\Validation\Validator` class at the top of the file:

```
/**
 * {@inheritdoc}
 */
protected function formatErrors(Validator $validator)
{
    return $validator->errors()->all();
}
```

Customizing The Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}
```

Manually Creating Validators

If you do not want to use the `ValidatesRequests` trait's `validate` method, you may create a validator instance manually using the `Validator` facade. The `make` method on the facade generates a new validator instance:

```
<?php
```

```
namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Store the blog post...
    }
}
```

The first argument passed to the `make` method is the data under validation. The second argument is the validation rules that should be applied to the data.

After checking if the request validation failed, you may use the `withErrors` method to flash the error messages to the session. When using this method, the `$errors` variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The `withErrors` method accepts a validator, a `MessageBag`, or a PHP `array`.

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the `ValidatesRequest` trait, you may call the `validate` method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an AJAX request, a

JSON response will be returned:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors, allowing you to retrieve the error messages for a specific form. Simply pass a name as the second argument to `withErrors`:

```
return redirect('register')
            ->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
{{ $errors->login->first('email') }}
```

After Validation Hook

The validator also allows you to attach callbacks to be run after validation is completed. This allows you to easily perform further validation and even add more error messages to the message collection. To get started, use the `after` method on a validator instance:

```
$validator = Validator::make(...);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

Working With Error Messages

After calling the `errors` method on a `Validator` instance, you will receive an `Illuminate\Support\MessageBag` instance, which has a variety of convenient methods for working with error messages. The `$errors` variable that is automatically made available to all views is also an instance of the `MessageBag` class.

Retrieving The First Error Message For A Field

To retrieve the first error message for a given field, use the `first` method:

```
$errors = $validator->errors();

echo $errors->first('email');
```

Retrieving All Error Messages For A Field

If you need to retrieve an array of all the messages for a given field, use the `get` method:

```
foreach ($errors->get('email') as $message) {
    //
}
```

If you are validating an array form field, you may retrieve all of the messages for each of the array elements using the `*` character:

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

Retrieving All Error Messages For All Fields

To retrieve an array of all messages for all fields, use the `all` method:

```
foreach ($errors->all() as $message) {
    //
}
```

Determining If Messages Exist For A Field

The `has` method may be used to determine if any error messages exist for a given field:

```
if ($errors->has('email')) {
    //
}
```

Custom Error Messages

If needed, you may use custom error messages for validation instead of the defaults. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

In this example, the `:attribute` place-holder will be replaced by the actual name of the field under validation. You may also utilize other place-holders in validation messages. For example:

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following types: :values',
];
```

Specifying A Custom Message For A Given Attribute

Sometimes you may wish to specify a custom error messages only for a specific field. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

Specifying Custom Messages In Language Files

In most cases, you will probably specify your custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

Specifying Custom Attributes In Language Files

If you would like the `:attribute` portion of your validation message to be replaced with a custom attribute name, you may specify the custom name in the `attributes` array of your `resources/lang/xx/validation.php` language file:

```
'attributes' => [
    'email' => 'email address',
],
```

Available Validation Rules

Below is a list of all available validation rules and their function:

Accepted Active URL After (Date) After Or Equal (Date) Alpha Alpha Dash Alpha Numeric Array Before (Date) Before Or Equal (Date) Between Boolean Confirmed Date Date Format Different Digits Digits Between Dimensions (Image Files) Distinct E-Mail Exists (Database) File Filled Image (File) In In Array Integer IP Address JSON Max MIME Types MIME Type By File Extension Min Nullable Not In Numeric Present Regular Expression Required Required If Required Unless Required With Required With All Required Without Required Without All Same Size String Timezone Unique (Database) URL

accepted

The field under validation must be *yes*, *on*, *1*, or *true*. This is useful for validating "Terms of Service" acceptance.

active_url

The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP function.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:*date*

The field under validation must be a value after or equal to the given date. For more information, see the after rule.

alpha

The field under validation must be entirely alphabetic characters.

alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

alpha_num

The field under validation must be entirely alpha-numeric characters.

array

The field under validation must be a PHP `array`.

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function.

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.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

date

The field under validation must be a valid date according to the `strtotime` PHP function.

date_format:*format*

The field under validation must match the given *format*. You should use **either** `date` or `date_format` when validating a field, not both.

different:*field*

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

digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

digits_between:*min*,*max*

The field under validation must have a length between the given *min* and *max*.

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 statement 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\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

distinct

When working with arrays, the field under validation must not have any duplicate values.

```
'foo.*.id' => 'distinct'
```

email

The field under validation must be formatted as an e-mail address.

exists:*table*,*column*

The field under validation must exist on a given database table.

Basic Usage Of Exists Rule

```
'state' => 'exists:states'
```

Specifying A Custom Column Name

```
'state' => 'exists:states,abbreviation'
```

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 using "dot" syntax:

```
'email' => 'exists:connection.staff,email'
```

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\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

file

The field under validation must be a successfully uploaded file.

filled

The field under validation must not be empty when it is present.

image

The file under validation must be an image (jpeg, png, bmp, gif, or svg)

in:*foo*,*bar*,...

The field under validation must be included in the given list of values. Since this rule often requires you to `implode` an array, the `Rule::in` method may be used to fluently construct the rule:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

in_array:*anotherfield*

The field under validation must exist in *anotherfield*'s values.

integer

The field under validation must be an integer.

ip

The field under validation must be an IP address.

ipv4

The field under validation must be an IPv4 address.

ipv6

The field under validation must be an IPv6 address.

json

The field under validation must be a valid JSON string.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

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 provided MIME type.

mimes:*foo,bar,...*

The file under validation must have a MIME type corresponding to one of the listed extensions.

Basic Usage Of MIME Rule

```
'photo' => 'mimes:jpeg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates against the MIME type of the file by reading the file's contents and guessing its MIME type.

A full listing of MIME types and their corresponding extensions may be found at the following location: https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

nullable

The field under validation may be `null`. This is particularly useful when validating primitive such as strings and integers that can contain `null` values.

not_in:*foo,bar,...*

The field under validation must not be included in the given list of values.

numeric

The field under validation must be numeric.

present

The field under validation must be present in the input data but can be empty.

regex:*pattern*

The field under validation must match the given regular expression.

Note: When using the `regex` pattern, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

required

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.

required_if:*anotherfield*,*value*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

required_unless:*anotherfield*,*value*,...

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*.

required_with:*foo*,*bar*,...

The field under validation must be present and not empty *only if* any of the other specified fields are present.

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.

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.

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.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value. For an array, *size* corresponds to the `count` of the array. For files, *size* corresponds to the file size in kilobytes.

string

The field under validation must be a string. If you would like to allow the field to also be `null`, you should assign the `nullable` rule to the field.

timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

unique:*table,column,except,idColumn*

The field under validation must be unique in a given database table. If the `column` option is not specified, the field name will be used.

Specifying A Custom Column Name:

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

Custom Database Connection

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

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

Forcing A Unique Rule To Ignore A Given ID:

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

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\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

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:

```
'email' => Rule::unique('users')->ignore($user->id, 'user_id')
```

Adding Additional Where Clauses:

You may also specify additional query constraints by customizing the query using the `where` method. For example, let's add a constraint that verifies the `account_id` is `1`:

```
'email' => Rule::unique('users')->where(function ($query) {
    $query->where('account_id', 1);
})
```

url

The field under validation must be a valid URL.

Conditionally Adding Rules

Validating When Present

In some situations, you may wish to run validation checks against a field **only** if that field is present in the input array. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

> {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

Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game resale shop, or maybe they just enjoy collecting. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
$v->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is the rules we want to add. If the `Closure` passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {
    return $input->games >= 100;
});
```

> {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.

Validating Arrays

Validating array based form input fields doesn't have to be a pain. For example, to validate that each e-mail 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 your validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ]
],
```

Custom Validation Rules

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;

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

```
    }

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

The custom validator Closure receives four arguments: the name of the `$attribute` being validated, the `$value` of the attribute, an array of `$parameters` passed to the rule, and the `Validator` instance.

You may also pass a class and method to the `extend` method instead of a Closure:

```
Validator::extend('foo', 'FooValidator@validate');
```

Defining The Error Message

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

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

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

// The rest of the validation error messages...
```

When creating a custom validation rule, you may sometimes need to define custom place-holder replacements for error messages. You may do so by creating a custom Validator as described above then making a call to the `replacer` method on the `Validator` facade. You may do this within the `boot` method of a service provider:

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

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

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

Implicit Extensions

By default, when an attribute being validated is not present or contains an empty value as defined by the `required` rule, normal validation rules, including custom extensions, are not run. For example, the `unique` rule will not be run against a `null` value:

```
$rules = ['name' => 'unique'];

$input = ['name' => null];

Validator::make($input, $rules)->passes(); // true
```

For a rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create such an "implicit" extension, use the `Validator::extendImplicit()` method:

```
Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

> {note} An "implicit" extension only *implies* that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Errors & Logging

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 triggered by your application are logged and then rendered back to the user. We'll dive deeper into this class throughout this documentation.

For logging, Laravel utilizes the Monolog library, which provides support for a variety of powerful log handlers. Laravel configures several of these handlers for you, allowing you to choose between a single log file, rotating log files, or writing error information to the system log.

Configuration

Error Detail

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 value is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.

Log Storage

Out of the box, Laravel supports writing log information to `single` files, `daily` files, the `syslog`, and the `errorlog`. To configure which storage mechanism Laravel uses, you should modify the `log` option in your `config/app.php` configuration file. For example, if you wish to use daily log files instead of a single file, you should set the `log` value in your `app` configuration file to `daily`:

```
'log' => 'daily'
```

Maximum Daily Log Files

When using the `daily` log mode, Laravel will only retain five days of log files by default. If you want to adjust the number of retained files, you may add a `log_max_files` configuration value to your `app` configuration file:

```
'log_max_files' => 30
```

Log Severity Levels

When using Monolog, log messages may have different levels of severity. By default, Laravel writes all log levels to storage. However, in your production environment, you may wish to configure the minimum severity that should be logged by adding the `log_level` option to your `app.php` configuration file.

Once this option has been configured, Laravel will log all levels greater than or equal to the specified severity. For example, a default `log_level` of `error` will log **error**, **critical**, **alert**, and **emergency** messages:

```
'log_level' => env('APP_LOG_LEVEL', 'error'),
```

> {tip} Monolog recognizes the following severity levels - from least severe to most severe: `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert`, `emergency`.

Custom Monolog Configuration

If you would like to have complete control over how Monolog is configured for your application, you may use the application's `configureMonologUsing` method. You should place a call to this method in your `bootstrap/app.php` file right before the `$app` variable is returned by the file:

```
$app->configureMonologUsing(function ($monolog) {
    $monolog->pushHandler(...);
});

return $app;
```

The Exception Handler

The Report Method

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`. We'll examine each of these methods in detail. The `report` method is used to log exceptions or send them to an external service like Bugsnag or Sentry. By default, the `report` method simply passes the exception to the base class where the exception is logged. However, you are free to log exceptions however you wish.

For example, if you need to report different types of exceptions in different ways, you may use the PHP `instanceof` comparison operator:

```
/**
 * Report or log an exception.
 *
```

```
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
     *
     * @param  \Exception  $exception
     * @return void
     */
    public function report(Exception $exception)
    {
        if ($exception instanceof CustomException) {
            //
        }

        return parent::report($exception);
    }
```

Ignoring Exceptions By Type

The `$dontReport` property of the exception handler contains an array of exception types that will not be logged. For example, exceptions resulting from 404 errors, as well as several other types of errors, are not written to your log files. You may add other exception types to this array as needed:

```
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */
    protected $dontReport = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
        \Symfony\Component\HttpKernel\Exception\HttpException::class,
        \Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Validation\ValidationException::class,
    ];
```

The Render Method

The `render` method is responsible for converting a given exception into an HTTP response that should be sent back to the browser. By default, the exception is passed to the base class which generates a response for you. However, you are free to check the exception type or return your own custom response:

```
    /**
     * Render an exception into an HTTP response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Exception  $exception
```

```
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $exception)
{
    if ($exception instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $exception);
}
```

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);
```

The `abort` helper will immediately raise an exception which will be rendered by the exception handler. Optionally, you may provide the response text:

```
abort(403, 'Unauthorized action.');
```

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 `HttpException` instance raised by the `abort` function will be passed to the view as an `$exception` variable:

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

Logging

Laravel provides a simple abstraction layer on top of the powerful Monolog library. By default, Laravel is configured to create a log file for your application in the `storage/logs` directory. You may write information to the logs using the `Log` facade:

```
<?php
```

```
namespace App\Http\Controllers;

use App\User;
use Illuminate\Support\Facades\Log;
use App\Http\Controllers\Controller;

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

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

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

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

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]);
```

Accessing The Underlying Monolog Instance

Monolog has a variety of additional handlers you may use for logging. If needed, you may access the underlying Monolog instance being used by Laravel:

```
$monolog = Log::getMonolog();
```

Blade Templates

Introduction

Blade is the simple, yet powerful templating engine provided with Laravel. Unlike other popular PHP templating engines, Blade does not restrict you from using plain PHP code in your views. In fact, all Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are *template inheritance* and *sections*. To get started, let's take a look at a simple example. First, we will examine a "master" 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:

```
<!-- Stored in 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 Blade `@extends` 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`:

```
<!-- Stored in 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.

Blade views may be returned from routes using the global `view` helper:

```
Route::get('blade', function () {
    return view('child');
});
```

Components & Slots

Components and slots provide similar benefits to sections and layouts; however, some may find the mental model of components and slots easier to understand. First, let's imagine a reusable "alert" component we would like to reuse throughout our application:

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

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

The `{{ $slot }}` variable will contain the content we wish to inject into the component. Now, to construct this component, we can use the `@component` Blade directive:

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Sometimes it is helpful to define multiple slots for a component. Let's modify our alert component to allow for the injection of a "title". Named slots may be displayed by simply "echoing" the variable that matches their name:

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

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

Now, we can inject content into the named slot using the `@slot` directive. Any content not within a `@slot` directive will be passed to the component in the `$slot` variable:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot
```

```
    You are not allowed to access this resource!
@endcomponent
```

Passing Additional Data To Components

Sometimes you may need to pass additional data to a component. For this reason, you can pass an array of data as the second argument to the `@component` directive. All of the data will be made available to the component template as variables:

```
@component('alert', ['foo' => 'bar'])
    ...
@endcomponent
```

Displaying Data

You may display data passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

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

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

Of course, 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() }}.
```

> {note} Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

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. Always 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 instead be rendered by your JavaScript framework.

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
```

Control Structures

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
```

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:

```
@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 condition with the directive declaration in one line:

```
@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).
`$loop->remaining`	The iteration remaining in the loop.
`$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.
`$loop->depth`	The nesting level of the current loop.
`$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 --}}
```

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
    //
@endphp
```

> {tip} While Blade provides this feature, using it frequently may be a signal that you have too much logic embedded within your template.

Including Sub-Views

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 extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

Of course, 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', ['some' => 'data'])
```

> {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 first argument is the view partial 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 your view partial. The key for the current iteration will be available as the `key` variable within your view partial.

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')
```

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>
```

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
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
```

```
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }

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

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.

Localization

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. Language strings are stored in files within the `resources/lang` directory. Within this directory there should be a subdirectory for each language supported by the application:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files simply return an array of keyed strings. For example:

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

Configuring The Locale

The default language for your application is stored in the `config/app.php` configuration file. Of course, you may modify this value to suit the needs of your application. You may also change the active language at runtime using the `setLocale` method on the `App` facade:

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($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 `getLocale` and `isLocale` methods on the `App` facade to determine the current locale or check if the locale is a given value:

```
$locale = App::getLocale();

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 the application:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files simply return an array of keyed strings. For example:

```
<?php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application'
];
```

Using Translation Strings As Keys

For applications with heavy translation requirements, defining every string with a "short key" can become quickly confusing when referencing them in your views. 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 la programación."
}
```

Retrieving Translation Strings

You may retrieve lines from language files using the `__` helper function. The `__` method accepts the file and key of the translation string as its first argument. For example, let's retrieve the `welcome` translation string from the `resources/lang/messages.php` language file:

```
echo __('messages.welcome');

echo __('I love programming.');
```

Of course if you are using the Blade templating engine, you may use the `{{ }}` syntax to echo the translation string or use the `@lang` directive:

```
{{ __('messages.welcome') }}

@lang('messages.welcome')
```

If the specified translation string does not exist, the `__` function will simply return the translation string key. So, using the example above, the `__` function would return `messages.welcome` if the translation string does not exist.

Replacing Parameters In Translation Strings

If you wish, you may define place-holders in your translation strings. All place-holders are prefixed with a `:`. For example, you may define a welcome message with a place-holder name:

```
'welcome' => 'Welcome, :name',
```

To replace the place-holders when retrieving a translation string, pass an array of replacements as the second argument to the `__` function:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

If your place-holder 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. By using a "pipe" character, you may distinguish singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

You may even create more complex pluralization rules which specify translation strings for multiple number ranges:

```
'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);
```

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.

JavaScript & CSS Scaffolding

- Introduction
- Writing CSS
- Writing JavaScript
  - Writing Vue Components

Introduction

While Laravel does not dictate which JavaScript or CSS pre-processors you use, it does provide a basic starting point using Bootstrap and Vue that will be helpful for many applications. By default, Laravel uses NPM to install both of these frontend packages.

CSS

Laravel Mix provides a clean, expressive API over compiling SASS or Less, which are extensions of plain CSS that add variables, mixins, and other powerful features that make working with CSS much more enjoyable. In this document, we will briefly discuss CSS compilation in general; however, you should consult the full Laravel Mix documentation for more information on compiling SASS or Less.

JavaScript

Laravel does not require you to use a specific JavaScript framework or library to build your applications. In fact, you don't have to use JavaScript at all. However, Laravel does include some basic scaffolding to make it easier to get started writing modern JavaScript using the Vue library. Vue provides an expressive API for building robust JavaScript applications using components. As with CSS, we may use Laravel Mix to easily compile JavaScript components into a single, browser-ready JavaScript file.

Writing CSS

Laravel's `package.json` file includes the `bootstrap-sass` package to help you get started prototyping your application's frontend using Bootstrap. However, feel free to add or remove packages from the `package.json` file as needed for your own application. You are not required to use the Bootstrap framework to build your Laravel application - it is simply provided as a good starting point for those who choose to use it.

Before compiling your CSS, install your project's frontend dependencies using the Node package manager (NPM):

```
npm install
```

Once the dependencies have been installed using `npm install`, you can compile your SASS files to plain CSS using Laravel Mix. The `npm run dev` command will process the instructions in your `webpack.mix.js` file. Typically, your compiled CSS will be placed in the `public/css` directory:

```
npm run dev
```

The default `webpack.mix.js` included with Laravel will compile the `resources/assets/sass/app.scss` SASS file. This `app.scss` file imports a file of SASS variables and loads Bootstrap, which provides a good starting point for most applications. Feel free to customize the `app.scss` file however you wish or even use an entirely different pre-processor by configuring Laravel Mix.

Writing JavaScript

All of the JavaScript dependencies required by your application can be found in the `package.json` file in the project's root directory. This file is similar to a `composer.json` file except it specifies JavaScript dependencies instead of PHP dependencies. You can install these dependencies using the Node package manager (NPM):

```
npm install
```

{tip} By default, the Laravel `package.json` file includes a few packages such as `vue` and `axios` to help you get started building your JavaScript application. Feel free to add or remove from the `package.json` file as needed for your own application.

Once the packages are installed, you can use the `npm run dev` command to compile your assets. Webpack is a module bundler for modern JavaScript applications. When you run the `npm run dev` command, Webpack will execute the instructions in your `webpack.mix.js` file:

```
npm run dev
```

By default, the Laravel `webpack.mix.js` file compiles your SASS and the `resources/assets/js/app.js` file. Within the `app.js` file you may register your Vue components or, if you prefer a different framework, configure your own JavaScript application. Your compiled JavaScript will typically be placed in the `public/js` directory.

{tip} The `app.js` file will load the `resources/assets/js/bootstrap.js` file which bootstraps and configures Vue, Axios, jQuery, and all other JavaScript dependencies. If you have additional JavaScript dependencies to configure, you may do so in this file.

Writing Vue Components

By default, fresh Laravel applications contain an `Example.vue` Vue component located in the `resources/assets/js/components` directory. The `Example.vue` file is an example of a single file Vue component which defines its JavaScript and HTML template in the same file. Single file components provide a very convenient approach to building JavaScript driven applications. The example component is registered in your `app.js` file:

```
Vue.component('example', require('./components/Example.vue'));
```

To use the component in your application, you may simply drop it into one of your HTML templates. For example, after running the `make:auth` Artisan command to scaffold your application's authentication and registration screens, you could drop the component into the `home.blade.php` Blade template:

```
@extends('layouts.app')

@section('content')
    <example></example>
@endsection
```

{tip} Remember, you should run the `npm run dev` command each time you change a Vue component. Or, you may run the `npm run watch` command to monitor and automatically recompile your components each time they are modified.

Of course, if you are interested in learning more about writing Vue components, you should read the Vue documentation, which provides a thorough, easy-to-read overview of the entire Vue framework.

Compiling Assets (Laravel Mix)

- Introduction
- Installation & Setup
- Running Mix
- Working With Stylesheets
  - Less
  - Sass
  - Stylus
  - PostCSS
  - Plain CSS
  - URL Processing
  - Source Maps
- Working With JavaScript
  - Vendor Extraction
  - React
  - Vanilla JS
  - Custom Webpack Configuration
- Copying Files & Directories
- Versioning / Cache Busting
- Browsersync Reloading
- Environment Variables
- Notifications

Introduction

Laravel Mix provides a fluent API for defining Webpack build steps for your Laravel application using several common CSS and JavaScript pre-processors. Through simple method chaining, you can fluently define your asset pipeline. For example:

```
mix.js('resources/assets/js/app.js', 'public/js')
   .sass('resources/assets/sass/app.scss', '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. Of course, you are free to use any asset pipeline tool you wish, or even none at all.

Installation & Setup

Installing Node

Before triggering Mix, you must first ensure that Node.js and NPM are installed on your machine.

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

By default, Laravel Homestead includes everything you need; however, if you aren't using Vagrant, then you can easily install the latest version of Node and NPM using simple graphical installers from their download page.

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 includes everything you need to get started. Think of this like your `composer.json` file, except it defines Node dependencies instead of PHP. You may install the dependencies it references by running:

```
npm install
```

If you are developing on a Windows system or you are running your VM on a Windows host system, you may need to run the `npm install` command with the `--no-bin-links` switch enabled:

```
npm install --no-bin-links
```

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 with the default Laravel `package.json` file:

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run production
```

Watching Assets For Changes

The `npm run watch` command will continue running in your terminal and watch all relevant files for changes. Webpack will then automatically recompile your assets when it detects a change:

```
npm run watch
```

You may find that in certain environments Webpack isn't updating when your files change. If this is the case on your system, consider using the `watch-poll` command:

```
npm run watch-poll
```

Working With Stylesheets

The `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.

Less

The `less` method may be used to compile Less into CSS. Let's compile our primary `app.less` file to `public/css/app.css`.

```
mix.less('resources/assets/less/app.less', 'public/css');
```

Multiple calls to the `less` method may be used to compile multiple files:

```
mix.less('resources/assets/less/app.less', 'public/css')
   .less('resources/assets/less/admin.less', 'public/css');
```

If you wish to customize the file name of the compiled CSS, you may pass a full file path as the second argument to the `less` method:

```
mix.less('resources/assets/less/app.less', 'public/stylesheets/styles.css');
```

If you need to override the underlying Less plug-in options, you may pass an object as the third argument to `mix.less()`:

```
mix.less('resources/assets/less/app.less', 'public/css', {
    strictMath: true
});
```

Sass

The `sass` method allows you to compile Sass into CSS. You may use the method like so:

```
mix.sass('resources/assets/sass/app.scss', 'public/css');
```

Again, like the `less` method, you may compile multiple Sass files into their own respective CSS files and even customize the output directory of the resulting CSS:

```
mix.sass('resources/assets/sass/app.sass', 'public/css')
   .sass('resources/assets/sass/admin.sass', 'public/css/admin');
```

Additional Node-Sass plug-in options may be provided as the third argument:

```
mix.sass('resources/assets/sass/app.sass', 'public/css', {
    precision: 5
});
```

Stylus

Similar to Less and Sass, the `stylus` method allows you to compile Stylus into CSS:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css');
```

You may also install additional Stylus plug-ins, such as Rupture. First, install the plug-in in question through NPM (`npm install rupture`) and then require it in your call to `mix.stylus()`:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css', {
    use: [
        require('rupture')()
    ]
});
```

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 plug-in to automatically apply all necessary CSS3 vendor prefixes. However, you're free to add any additional plug-ins that are appropriate for your application. First, install the desired plug-in through NPM and then reference it in your `webpack.mix.js` file:

```
mix.sass('resources/assets/sass/app.scss', 'public/css')
   .options({
        postCss: [
            require('postcss-css-variables')()
        ]
   });
```

Plain CSS

If you would just like to concatenate some plain CSS stylesheets into a single file, you may use the `styles` method.

```
mix.styles([
    'public/css/vendor/normalize.css',
    'public/css/vendor/videojs.css'
], 'public/css/all.css');
```

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 `url()`s 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/assets/app/app.scss', 'public/css')
   .options({
      processCssUrls: false
   });
```

With this addition to your `webpack.mix.js` file, Mix will no longer match `url()`s 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/assets/js/app.js', 'public/js')
   .sourceMaps();
```

Working With JavaScript

Mix provides several features to help you work with your JavaScript files, such as compiling ECMAScript 2015, module bundling, minification, and simply concatenating plain JavaScript files. Even better, this all works seamlessly, without requiring an ounce of custom configuration:

```
mix.js('resources/assets/js/app.js', 'public/js');
```

With this single line of code, you may now take advantage of:

- ES2015 syntax.
- Modules
- Compilation of `.vue` files.
- Minification for production environments.

Vendor Extraction

One potential downside to bundling all application-specific JavaScript with your vendor libraries 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/assets/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 above snippet 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>
```

React

Mix can automatically install the Babel plug-ins necessary for React support. To get started, replace your `mix.js()` call with `mix.react()`:

```
mix.react('resources/assets/js/app.jsx', 'public/js');
```

Behind the scenes, Mix will download and include the appropriate `babel-preset-react` Babel plug-in.

Vanilla JS

Similar to combining stylesheets with `mix.styles()`, you may also combine and minify any number of JavaScript files with the `scripts()` method:

```
mix.scripts([
    'public/js/admin.js',
    'public/js/dashboard.js'
], 'public/js/all.js');
```

This option is particularly useful for legacy projects where you don't require Webpack compilation for your JavaScript.

> {tip} A slight variation of `mix.scripts()` is `mix.babel()`. Its method signature is identical to `scripts`; however, the concatenated file will receive Babel compilation, which translates any ES2015 code to vanilla JavaScript that all browsers will understand.

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. You might have a special loader or plug-in that needs to be referenced, or maybe you prefer to use Stylus instead of Sass. 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 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.

Copying Files & Directories

The `copy` method may be used to copy files and directories to new locations. This can be useful when a particular asset within your `node_modules` directory needs to be relocated to your `public` folder.

```
mix.copy('node_modules/foo/bar.css', 'public/css/bar.css');
```

When copying a directory, the `copy` method will flatten the directory's structure. To maintain the directory's original structure, you should use the `copyDirectory` method instead:

```
mix.copyDirectory('assets/img', 'public/img');
```

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 handle this for you using the `version` method.

The `version` method will automatically append a unique hash to the filenames of all compiled files, allowing for more convenient cache busting:

```
mix.js('resources/assets/js/app.js', 'public/js')
   .version();
```

After generating the versioned file, you won't know the exact file name. 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:

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}">
```

Because versioned files are usually unnecessary in development, you may wish to instruct the versioning process to only run during `npm run production`:

```
mix.js('resources/assets/js/app.js', 'public/js');

if (mix.config.inProduction) {
    mix.version();
}
```

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 by calling the `mix.browserSync()` method:

```
mix.browserSync('my-domain.dev');

// Or...

// https://browsersync.io/docs/options
mix.browserSync({
    proxy: 'my-domain.dev'
});
```

You may pass either a string (proxy) or object (BrowserSync settings) to this method. Next, start Webpack's dev server using the `npm run watch` command. Now, when you modify a script or PHP file, watch as the browser instantly refreshes the page to reflect your changes.

Environment Variables

You may inject environment variables into Mix by prefixing a key 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 via the `process.env` object. If the value changes while you are running a `watch` task, you will need to restart the task:

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

Notifications

When available, Mix will automatically display OS notifications for each bundle. This will give you instant feedback, as to whether the compilation was successful or not. However, there may be instances when you'd prefer to disable these notifications. One such example might be triggering Mix on your production server. Notifications may be deactivated, via the `disableNotifications` method.

```
mix.disableNotifications();
```

Authentication

Introduction

> {tip} **Want to get started fast?** Just run `php artisan make:auth` and `php artisan migrate` in a fresh Laravel application. Then, navigate your browser to `http://your-app.dev/register` or any other URL that is assigned to your application. These two commands will take care of scaffolding your entire authentication system!

Laravel makes implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file

is located at `config/auth.php`, which contains several well documented options for tweaking the behavior of the authentication services.

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.

Don't worry if this all sounds confusing now! Many applications will never need to modify the default authentication configuration.

Database Considerations

By default, Laravel includes an `App\User` Eloquent model in your `app` 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 driver which uses the Laravel query builder.

When building the database schema for the `App\User` model, make sure the password column is at least 60 characters in length. Maintaining the default string column length of 255 characters would be a good choice.

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.

Authentication Quickstart

Laravel ships with several pre-built authentication controllers, which are located in the `App\Http\Controllers\Auth` namespace. The `RegisterController` handles new user registration, the `LoginController` handles authentication, the `ForgotPasswordController` handles e-mailing links for resetting passwords, and the `ResetPasswordController` contains the logic to reset passwords. Each of these controllers uses a trait to include their necessary methods. For many applications, you will not need to modify these controllers at all.

Routing

Laravel provides a quick way to scaffold all of the routes and views you need for authentication using one simple command:

```
php artisan make:auth
```

This command should be used on fresh applications and will install a layout view, registration and login views, as well as routes for all authentication end-points. A `HomeController` will also be generated to handle post-login requests to your application's dashboard.

Views

As mentioned in the previous section, the `php artisan make:auth` command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.

The `make:auth` command will also create a `resources/views/layouts` directory containing a base layout for your application. All of these views use the Bootstrap CSS framework, but you are free to customize them however you wish.

Authenticating

Now that you have routes and views setup for the included authentication controllers, you are ready to register and authenticate new users for your application! You may simply access your application in a browser since the authentication controllers already contain the logic (via their traits) to authenticate existing users and store new users in the database.

Path Customization

When a user is successfully authenticated, they will be redirected to the `/home` URI. You can customize the post-authentication redirect location by defining a `redirectTo` property on the `LoginController`, `RegisterController`, and `ResetPasswordController`:

```
protected $redirectTo = '/';
```

If the redirect path needs custom generation logic you may define a `redirectTo` method instead of a `redirectTo` property:

```
protected function redirectTo()
{
    return '/path';
}
```

{tip} The `redirectTo` method will take precedence over the `redirectTo` attribute.

Username Customization

By default, Laravel uses the `email` field for authentication. If you would like to customize this, you may define a `username` method on your `LoginController`:

```
public function username()
{
    return 'username';
}
```

Guard Customization

You may also customize the "guard" that is used to authenticate and register users. To get started, define a `guard` method on your `LoginController`, `RegisterController`, and `ResetPasswordController`. The method should return a guard instance:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Validation / Storage Customization

To modify the form fields that are required when a new user registers with your application, or to customize how new users are stored into your database, you may modify the `RegisterController` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `RegisterController` contains the validation rules for new users of the application. You are free to modify this method as you wish.

The `create` method of the `RegisterController` is responsible for creating new `App\User` records in your database using the Eloquent ORM. You are free to modify this method according to the needs of your database.

Retrieving The Authenticated User

You may access the authenticated user via the `Auth` facade:

```
use Illuminate\Support\Facades\Auth;

// Get the currently authenticated user...
$user = Auth::user();
```

```
// Get 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:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller
{
    /**
     * Update the user's profile.
     *
     * @param  Request  $request
     * @return Response
     */
    public function update(Request $request)
    {
        // $request->user() returns an instance of the authenticated user...
    }
}
```

Determining If The Current User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method on the `Auth` facade, which 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 is defined at `Illuminate\Auth\Middleware\Authenticate`. Since this middleware is already registered in your HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('profile', function () {
    // Only authenticated users may enter...
})->middleware('auth');
```

Of course, if you are using controllers, you may call the `middleware` method from the controller's constructor instead of attaching it in the route definition directly:

```
public function __construct()
{
    $this->middleware('auth');
}
```

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:

```
public function __construct()
{
    $this->middleware('auth:api');
}
```

Login Throttling

If you are using Laravel's built-in `LoginController` class, the `Illuminate\Foundation\Auth\ThrottlesLogins` trait will already be included in your controller. 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 / e-mail address and their IP address.

Manually Authenticating Users

Of course, you are not required to use the authentication controllers included with Laravel. If you choose to remove these controllers, 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:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password])) {
            // Authentication passed...
            return redirect()->intended('dashboard');
        }
    }
}
```

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 hashed `password` value passed to the method via the array. If the two hashed passwords match an authenticated session will be started for the user.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

The `intended` method on the 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 conditions to the authentication query in addition to the user's e-mail and password. For example, we may verify that user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // The user is active, not suspended, and exists.
}
```

{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.

Accessing Specific Guard Instances

You may specify which guard instance you would like to utilize using the `guard` method on the `Auth` facade. 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)) {
    //
}
```

Logging Out

To log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

```
Auth::logout();
```

Remembering Users

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, which will keep the user authenticated indefinitely, or until they manually logout. Of course, your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token.

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

{tip} If you are using the built-in `LoginController` that is shipped with Laravel, the proper logic to "remember" users is already implemented by the traits used by the controller.

If you are "remembering" users, you may use the `viaRemember` method to determine if the user was authenticated using the "remember me" cookie:

```
if (Auth::viaRemember()) {
    //
}
```

Other Authentication Methods

Authenticate A User Instance

If you need to log an existing user instance into your application, you may call the `login` method with the user instance. The given object must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` contract. Of course, the `App\User` model included with Laravel already implements this interface:

```
Auth::login($user);

// Login and "remember" the given user...
Auth::login($user, true);
```

Of course, you may specify the guard instance you would like to use:

```
Auth::guard('admin')->login($user);
```

Authenticate A User By ID

To log a user into the application by their ID, you may use the `loginUsingId` method. This method simply accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);

// Login and "remember" the given user...
Auth::loginUsingId(1, true);
```

Authenticate A User Once

You may use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized, which means this method may be helpful when building a stateless API:

```
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 your 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 enter...
})->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 use the `email` column on the user record as the "username".

A Note On FastCGI

If you are using PHP FastCGI, HTTP Basic authentication may not work correctly out of the box. The following lines should be added to your `.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, which is particularly useful for API authentication. To do so, 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 Illuminate\Auth\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 enter...
})->middleware('auth.basic.once');
```

Adding Custom Guards

You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place this call to `provider` 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\Support\Facades\Auth;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

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 use this guard in the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

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:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::provider('riak', function ($app, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\UserProvider...

            return new RiakUserProvider($app->make('riak.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' => 'riak',
    ],
],
```

Finally, you may use this provider in your `guards` configuration:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

The User Provider Contract

The `Illuminate\Contracts\Auth\UserProvider` implementations are only responsible for fetching a `Illuminate\Contracts\Auth\Authenticatable` implementation out of a persistent storage system, such as MySQL, Riak, 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 it.

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`, stored in a field `remember_token`. As with the previous method, the `Authenticatable` implementation should be returned.

The `updateRememberToken` method updates the `$user` field `remember_token` with the new `$token`. The new token can be either a fresh token, assigned on a successful "remember me" login attempt, or `null` when the user is logging out.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to sign into 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 on `$credentials['username']`. The method should then 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 should probably use `Hash::check` to compare the value of `$user->getAuthPassword()` to the value of `$credentials['password']`. This method should return `true` or `false` indicating on 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, the provider should return implementations of this interface from the `retrieveById` 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. In a MySQL back-end, again, this would be the auto-incrementing primary key. The `getAuthPassword` 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 `User` class in the `app` directory which implements this interface, so you may consult this class for an implementation example.

Events

Laravel raises 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\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
```

```
    ],
];
```

API Authentication (Passport)

Introduction

Laravel already makes it easy to perform authentication via traditional login forms, but what about APIs? APIs typically use tokens to authenticate users and do not maintain session state between requests. Laravel makes API authentication a breeze using Laravel Passport, which 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 Alex Bilbie.

{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.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Next, register the Passport service provider in the `providers` array of your `config/app.php` configuration file:

```
Laravel\Passport\PassportServiceProvider::class,
```

The Passport service provider registers its own database migration directory with the framework, so you should migrate your database after registering the provider. The Passport migrations will create the tables your application needs to store clients and access tokens:

```
php artisan migrate
```

{note} If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `AppServiceProvider`. You may export the default migrations using `php artisan vendor:publish --tag=passport-migrations`.

Next, you should run the `passport:install` 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
```

After running this command, add the `Laravel\Passport\HasApiTokens` trait to your `App\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;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
```

```
    use HasApiTokens, Notifiable;
}
```

Next, you should call the `Passport::routes` method within the `boot` method of your `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 Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, in your `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',
    ],
],
```

Frontend Quickstart

> {note} In order to use the Passport Vue components, you must be using the Vue JavaScript framework. These components also use the Bootstrap CSS framework. However, even if you are not using these tools, the components serve as a valuable reference for your own frontend implementation.

Passport ships with a JSON API that you may use to allow your users to create clients and personal access tokens. However, it can be time consuming to code a frontend to interact with these APIs. So, Passport also includes pre-built Vue components you may use as an example implementation or starting point for your own implementation.

To publish the Passport Vue components, use the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-components
```

The published components will be placed in your `resources/assets/js/components` directory. Once the components have been published, you should register them in your `resources/assets/js/app.js` file:

```
Vue.component(
    'passport-clients',
    require('./components/passport/Clients.vue')
);

Vue.component(
    'passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
    'passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue')
);
```

After registering the components, make sure to run `npm run dev` to recompile your assets. Once you have recompiled your assets, you may drop the components

into one of your application's templates to get started creating clients and personal access tokens:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Deploying Passport

When deploying Passport to your production 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 token. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

Configuration

Token Lifetimes

By default, Passport issues long-lived access tokens that never need to be refreshed. If you would like to configure a shorter token lifetime, you may use the `tokensExpireIn` and `refreshTokensExpireIn` methods. These methods should be called from the `boot` method of your `AuthServiceProvider`:

```
use Carbon\Carbon;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(Carbon::now()->addDays(15));

    Passport::refreshTokensExpireIn(Carbon::now()->addDays(30));
}
```

Issuing Access Tokens

Using OAuth2 with 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
```

JSON API

Since your 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.

> {tip} If you don't want to implement the entire client management frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`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:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://your-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 `resources/views/vendor/passport`:

```
php artisan vendor:publish --tag=passport-views
```

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 then 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. In this example, we'll use the Guzzle HTTP library to make the `POST` request:

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);

    return json_decode((string) $response->getBody(), true);
});
```

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.

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. In this example, we'll use the Guzzle HTTP library to refresh the token:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
```

```
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

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.

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 e-mail 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` 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:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
```

```
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

> {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, 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` grant:

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
]);
```

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 your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
```

```
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Once a 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:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-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. To retrieve a token, make a request to the `oauth/token` endpoint:

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

echo json_decode((string) $response->getBody(), true);
```

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.

> {note} Personal access tokens are always long-lived. Their lifetime is not modified when using the `tokensExpireIn` or `refreshTokensExpireIn` methods.

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 using the `passport:client` 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
```

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 `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:

```
$user = App\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.

{tip} If you don't want to implement the personal access token frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`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 token so that they may edit or delete 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 delete 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 require a valid access token:

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

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:

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);
```

Token Scopes

Defining 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.

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your `AuthServiceProvider`. 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:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

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://your-app.com/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `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('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('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 `User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('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:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

This Passport 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. Now, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/user')
    .then(response => {
        console.log(response.data);
    });
```

When using this method of authentication, Axios will automatically send the `X-CSRF-TOKEN` header. In addition, the default Laravel JavaScript scaffolding instructs Axios to send the `X-Requested-With` header:

```
window.axios.defaults.headers.common = {
    'X-Requested-With': 'XMLHttpRequest',
};
```

> {note} If you are using a different JavaScript framework, you should make sure it is configured to send the `X-CSRF-TOKEN` and `X-Requested-With` headers with every outgoing request.

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. You may attach listeners to these events in your application's `EventServiceProvider`:

“‘php /** * 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:

```
public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```

Authorization

Introduction

In addition to providing authentication services out of the box, Laravel also provides a simple way to authorize user actions against a given resource. Like authentication, Laravel's approach to authorization is simple, and there are 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

their logic around a particular model or resource. 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 a 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

Gates are Closures that determine if a user is authorized to perform a given action and are typically defined in 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:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function ($user, $post) {
        return $user->id == $post->user_id;
    });
}
```

Authorizing Actions

To authorize an action using gates, you should use the `allows` or `denies` methods. 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:

```
if (Gate::allows('update-post', $post)) {
    // The current user can update the post...
}
```

```
if (Gate::denies('update-post', $post)) {
    // The current user can't update the post...
}
```

If you would like to determine if a particular 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...
}
```

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 `Post` model and a corresponding `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 the basic "CRUD" policy methods already included in the class, you may specify a `--model` when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

> {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.

Registering Policies

Once the policy exists, it needs to be registered. The `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 model:

```php
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

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();

        //
    }
}
```

Writing Policies

Policy Methods

Once the policy 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 `User` can update a given `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, for this example, let's verify that the user's `id` matches the `user_id` on the post:

```
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\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` actions, but remember you are free to give your policy methods any name you like.

> {tip} If you used the `--model` option when generating your policy via the Artisan console, it will already contain methods for the `view`, `create`, `update`, and `delete` actions.

Methods Without Models

Some policy methods only receive the currently authenticated user and not an instance of the model they authorize. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to check if a user is authorized to create any posts at all.

When defining policy methods that will not receive a model instance, such as a `create` method, it will not receive a model instance. Instead, you should define the method as only expecting the authenticated user:

```
/**
 * Determine if the given user can create posts.
 *
 * @param  \App\User  $user
 * @return bool
```

```
 */
public function create(User $user)
{
    //
}
```

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:

```
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

If you would like to deny all authorizations for a user you should return `false` from the `before` method. If `null` is returned, the authorization will fall through to the policy method.

Authorizing Actions Using Policies

Via The User Model

The `User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cant`. The `can` method receives the action you wish to authorize and the relevant model. For example, let's determine if a user is authorized to update a given `Post` model:

```
if ($user->can('update', $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 like `create` may 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:

```
use App\Post;

if ($user->can('create', Post::class)) {
    // Executes the "create" method on the relevant policy...
}
```

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 blog post:

```
use App\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 `Post` model will be passed to the policy method. If the user is not authorized to perform the given action, a HTTP response with a `403` status code will be generated by the middleware.

Actions That Don't Require Models

Again, some actions like `create` may 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\Post');
```

Via Controller Helpers

In addition to helpful methods provided to the `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, which the default Laravel exception handler will convert to an HTTP response with a 403 status code:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param  Request  $request
     * @param  Post  $post
     * @return Response
     */
    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 actions like create may not require a model instance. In these situations, you may pass a class name to the authorize method. The class name will be used to determine which policy to use when authorizing the action:

```
/**
 * Create a new blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function create(Request $request)
```

```
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}
```

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` family of directives:

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@elsecan('create', $post)
    <!-- The Current User Can Create New Post -->
@endcan

@cannot('update', $post)
    <!-- The Current User Can't Update The Post -->
@elsecannot('create', $post)
    <!-- The Current User Can't Create New Post -->
@endcannot
```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above respectively translate 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 Can't Update The Post -->
@endunless
```

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\Post::class)
    <!-- The Current User Can Create Posts -->
@endcan
```

```
@cannot('create', App\Post::class)
    <!-- The Current User Can't Create Posts -->
@endcannot
```

Encryption

- Introduction
- Configuration
- Using The Encrypter

Introduction

Laravel's encrypter uses OpenSSL to provide AES-256 and AES-128 encryption. You are strongly encouraged to use Laravel's built-in encryption facilities and not attempt to roll your own "home grown" encryption algorithms. All of Laravel's encrypted values are signed using a message authentication code (MAC) so that their underlying value can not be modified once encrypted.

Configuration

Before using Laravel's encrypter, you must set a `key` option in your `config/app.php` configuration file. You should use the `php artisan key:generate` command to generate this key since this Artisan command will use PHP's secure random bytes generator to build your key. If this value is not properly set, all values encrypted by Laravel will be insecure.

Using The Encrypter

Encrypting A Value

You may encrypt a value using the `encrypt` helper. All encrypted values are encrypted using OpenSSL and the `AES-256-CBC` cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC) to detect any modifications to the encrypted string:

```
<?php

namespace App\Http\Controllers;

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

```
class UserController extends Controller
{
    /**
     * Store a secret message for the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => encrypt($request->secret)
        ])->save();
    }
}
```

Encrypting Without Serialization

Encrypted values are passed through `serialize` during encryption, which allows for encryption of objects and arrays. Thus, non-PHP clients receiving encrypted values will need to `unserialize` the data. If you would like to encrypt and decrypt values without serialization, you may use the `encryptString` and `decryptString` methods of the `Crypt` facade:

```
use Illuminate\Support\Facades\Crypt;

$encrypted = Crypt::encryptString('Hello world.');

$decrypted = Crypt::decryptString($encrypted);
```

Decrypting A Value

You may decrypt values using the `decrypt` helper. If the value can not be properly decrypted, such as when the MAC is invalid, an `Illuminate\Contracts\Encryption\DecryptException` will be thrown:

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
```

```
}
```

Hashing

- Introduction
- Basic Usage

Introduction

The Laravel `Hash` facade provides secure Bcrypt hashing for storing user passwords. If you are using the built-in `LoginController` and `RegisterController` classes that are included with your Laravel application, they will automatically use Bcrypt for registration and authentication.

> {tip} 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.

Basic Usage

You may hash a password by calling the `make` method on the `Hash` facade:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Http\Controllers\Controller;

class UpdatePasswordController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function update(Request $request)
    {
        // Validate the new password length...

        $request->user()->fill([
```

```
            'password' => Hash::make($request->newPassword)
        ])->save();
    }
}
```

Verifying A Password Against A Hash

The `check` method allows you to verify that a given plain-text string corresponds to a given hash. However, if you are using the `LoginController` included with Laravel, you will probably not need to use this directly, as this controller automatically calls this method:

```
if (Hash::check('plain-text', $hashedPassword)) {
    // The passwords match...
}
```

Checking If A Password Needs To Be Rehashed

The `needsRehash` function allows you to determine if the work factor used by the hasher has changed since the password was hashed:

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

Resetting Passwords

- Introduction
- Database Considerations
- Routing
- Views
- After Resetting Passwords
- Customization

Introduction

> {tip} **Want to get started fast?** Just run `php artisan make:auth` in a fresh Laravel application and navigate your browser to `http://your-app.dev/register` or any other URL that is assigned to your application. This single command will take care of scaffolding your entire authentication system, including resetting passwords!

Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this on each application, Laravel provides convenient methods for sending password reminders and performing password resets.

> {note} Before using the password reset features of Laravel, your user must use the `Illuminate\Notifications\Notifiable` trait.

Database Considerations

To get started, verify that your `App\User` model implements the `Illuminate\Contracts\Auth\CanResetPasswo` contract. Of course, the `App\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.

Generating The Reset Token Table Migration

Next, a table must be created to store the password reset tokens. The migration for this table is included with Laravel out of the box, and resides in the `database/migrations` directory. So, all you need to do is run your database migrations:

```
php artisan migrate
```

Routing

Laravel includes `Auth\ForgotPasswordController` and `Auth\ResetPasswordController` classes that contains the logic necessary to e-mail password reset links and reset user passwords. All of the routes needed to perform password resets may be generated using the `make:auth` Artisan command:

```
php artisan make:auth
```

Views

Again, Laravel will generate all of the necessary views for password reset when the `make:auth` command is executed. These views are placed in `resources/views/auth/passwords`. You are free to customize them as needed for your application.

After Resetting Passwords

Once you have defined the routes and views to reset your user's passwords, you may simply access the route in your browser at `/password/reset`.

The `ForgotPasswordController` included with the framework already includes the logic to send the password reset link e-mails, while the `ResetPasswordController` includes the logic to reset user passwords.

After a password is reset, the user will automatically be logged into the application and redirected to `/home`. You can customize the post password reset redirect location by defining a `redirectTo` property on the `ResetPasswordController`:

```
protected $redirectTo = '/dashboard';
```

> {note} By default, password reset tokens expire after one hour. You may change this via the password reset `expire` option in your `config/auth.php` file.

Customization

Authentication Guard Customization

In your `auth.php` configuration file, you may configure multiple "guards", which may be used to define authentication behavior for multiple user tables. You can customize the included `ResetPasswordController` to use the guard of your choice by overriding the `guard` method on the controller. This method should return a guard instance:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Password Broker Customization

In your `auth.php` configuration file, you may configure multiple password "brokers", which may be used to reset passwords on multiple user tables. You can customize the included `ForgotPasswordController` and `ResetPasswordController` to use the broker of your choice by overriding the `broker` method:

```
use Illuminate\Support\Facades\Password;

/**
 * Get the broker to be used during password reset.
 *
 * @return PasswordBroker
 */
protected function broker()
{
```

```
    return Password::broker('name');
}
```

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 `User` model. Within this method, you may send the notification using any notification class you choose. The password reset `$token` is the first argument received by the method:

```
/**
 * Send the password reset notification.
 *
 * @param  string  $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $this->notify(new ResetPasswordNotification($token));
}
```

Artisan Console

Introduction

Artisan is the command-line interface included with Laravel. It 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, simply precede the name of the command with `help`:

```
php artisan help migrate
```

Laravel REPL

All Laravel applications include Tinker, a REPL powered by the PsySH package. Tinker allows you to interact with your entire Laravel application on the command line, including the Eloquent ORM, jobs, events, and more. To enter the Tinker environment, run the `tinker` Artisan command:

```
php artisan tinker
```

Writing Commands

In addition to the commands provided with Artisan, you may also 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, 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, since it will be created the first time you run the `make:command` Artisan command. The generated command will include the default set of properties and methods that are present on all commands:

```
php artisan make:command SendEmails
```

Next, you will need to register the command before it can be executed via the Artisan CLI.

Command Structure

After generating your command, you should fill in the `signature` and `description` properties of the class, which will be used when displaying your command on the `list` screen. The `handle` method will be called when your command is executed. You may place your command logic in this method.

> {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 below, note that we inject a service class to do the "heavy lifting" of sending the e-mails.

Let's take a look at an example command. Note that we are able to inject any dependencies we need into the command's constructor. The Laravel service container will automatically inject all dependencies type-hinted in the constructor:

```
<?php

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'email:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;
```

```
    /**
     * Create a new command instance.
     *
     * @param  DripEmailer  $drip
     * @return void
     */
    public function __construct(DripEmailer $drip)
    {
        parent::__construct();

        $this->drip = $drip;
    }

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}
```

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 routes 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('build {project}', function ($project) {
    $this->info("Building {$project}!");
});
```

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\User;
use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

Closure Command Descriptions

When defining a Closure based command, you may use the `describe` 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('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

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 = 'email:send {user}';
```

You may also make arguments optional and define default values for arguments:

```
// Optional argument...
email:send {user?}

// Optional argument with default value...
email:send {user=foo}
```

Options

Options, like arguments, are another form of user input. Options are prefixed by two hyphens (`--`) when they are specified on 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 = 'email: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 email: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, suffix the option name with a `=` sign:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue=}';
```

In this example, the user may pass a value for the option like so:

```
php artisan email: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:

```
email:send {user} {--queue=default}
```

Option Shortcuts

To assign a shortcut when defining an option, you may specify it before the option name and use a | delimiter to separate the shortcut from the full option name:

```
email:send {user} {--Q|queue}
```

Input Arrays

If you would like to define arguments or options to expect array inputs, you may use the `*` character. First, let's take a look at an example that specifies an array argument:

```
email: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 `['foo', 'bar']`:

```
php artisan email:send foo bar
```

When defining an option that expects an array input, each option value passed to the command should be prefixed with the option name:

```
email:send {user} {--id=*}

php artisan email:send --id=1 --id=2
```

Input Descriptions

You may assign descriptions to input arguments and options by separating the parameter 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 = 'email: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 obviously 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:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
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...
$options = $this->options();
```

If the argument or option does not exist, `null` will be returned.

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 a password:

```
$password = $this->secret('What is the password?');
```

Asking For Confirmation

If you need to ask the user for a simple 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?')) {
    //
}
```

Auto-Completion

The `anticipate` method can be used to provide auto-completion for possible choices. The user can still choose any answer, regardless of the auto-completion hints:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Multiple Choice Questions

If you need to give the user a predefined set of choices, you may use the `choice` method. You may set the default value to be returned if no option is chosen:

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $default);
```

Writing Output

To send output to the console, 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 text:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->info('Display this on the screen');
}
```

To display an error message, use the `error` method. Error message text is typically displayed in red:

```
$this->error('Something went wrong!');
```

If you would like to display plain, uncolored console output, use the `line` method:

```
$this->line('Display this on the screen');
```

Table Layouts

The `table` method makes it easy to correctly format multiple rows / columns of data. Just pass in the headers and rows to the method. The width and height will be dynamically calculated based on the given data:

```
$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);
```

Progress Bars

For long running tasks, it could be helpful to show a progress indicator. Using the output object, we can start, advance and stop the Progress Bar. First, define the total number of steps the process will iterate through. Then, advance the Progress Bar after processing each item:

```
$users = App\User::all();

$bar = $this->output->createProgressBar(count($users));

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}
```

```
$bar->finish();
```

For more advanced options, check out the Symfony Progress Bar component documentation.

Registering Commands

Once your command is finished, you need to register it with Artisan. All commands are registered in the `app/Console/Kernel.php` file. Within this file, you will find a list of commands in the `commands` property. To register your command, simply add the command's class name to the list. 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 fire 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 the name of the command as the first argument, and an array of command parameters as the second argument. The exit code will be returned:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

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:

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
```

```
});
```

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 may pass `true` or `false`:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

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 parameters:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
}
```

If you would like to call another console command and suppress all of its output, you may use the `callSilent` method. The `callSilent` method has the same signature as the `call` method:

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Broadcasting

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. This provides a more robust, efficient alternative to continually polling your application for changes.

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

> {tip} Before diving into event broadcasting, make sure you have read all of the documentation regarding Laravel events and listeners.

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, Redis, and a `log` driver for local development and debugging. Additionally, a `null` driver is included which allows you to totally disable broadcasting. 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 fresh Laravel applications, you only need to uncomment this provider in the `providers` array of your `config/app.php` configuration file. This provider will allow you to register the broadcast authorization routes and callbacks.

CSRF Token

Laravel Echo will need access to the current session's CSRF token. If available, Echo will pull the token from the `Laravel.csrfToken` JavaScript object. This object is defined in the `resources/views/layouts/app.blade.php` layout that is created if you run the `make:auth` Artisan command. If you are not using this layout, you may define a `meta` tag in your application's `head` HTML element:

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

Driver Prerequisites

Pusher

If you are broadcasting your events over Pusher, you should install the Pusher PHP SDK using the Composer package manager:

```
composer require pusher/pusher-php-server
```

Next, you should configure your Pusher credentials in the `config/broadcasting.php` configuration file. An example Pusher configuration is already included in this file, allowing you to quickly specify your Pusher key, secret, and application ID. The `config/broadcasting.php` file's `pusher` configuration also allows you to specify additional `options` that are supported by Pusher, such as the cluster:

```
'options' => [
    'cluster' => 'eu',
    'encrypted' => true
],
```

When using Pusher and Laravel Echo, you should specify `pusher` as your desired broadcaster when instantiating the Echo instance in your `resources/assets/js/bootstrap.js` file:

```
import Echo from "laravel-echo"

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
```

```
});
```

Redis

If you are using the Redis broadcaster, you should install the Predis library:

```
composer require predis/predis
```

The Redis broadcaster will broadcast messages using Redis' pub / sub feature; however, you will need to pair this with a WebSocket server that can receive the messages from Redis and broadcast them to your WebSocket channels.

When the Redis broadcaster publishes an event, it will be published on the event's specified channel names and the payload will be a JSON encoded string containing the event name, a `data` payload, and the user that generated the event's socket ID (if applicable).

Socket.IO

If you are going to pair the Redis broadcaster with a Socket.IO server, you will need to include the Socket.IO JavaScript client library in your application's `head` HTML element. When the Socket.IO server is started, it will automatically expose the client JavaScript library at a standard URL. For example, if you are running the Socket.IO server on the same domain as your web application, you may access the client library like so:

```
<script src="//{{ Request::getHost() }}:6001/socket.io/socket.io.js"></script>
```

Next, you will need to instantiate Echo with the `socket.io` connector and a `host`.

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'socket.io',
    host: window.location.hostname + ':6001'
});
```

Finally, you will need to run a compatible Socket.IO server. Laravel does not include a Socket.IO server implementation; however, a community driven Socket.IO server is currently maintained at the tlaverdure/laravel-echo-server GitHub repository.

Queue Prerequisites

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

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 and Redis 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.

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. We won't discuss the details of configuring Pusher or Laravel Echo since that will be discussed in detail in other sections of this documentation.

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 `ShippingStatusUpdated` event is fired when a shipping status update is processed by the application:

```
event(new ShippingStatusUpdated($update));
```

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 `ShippingStatusUpdated` event with the `ShouldBroadcast` interface. This will instruct Laravel to broadcast the event when it is fired:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast
{
```

```
    /**
     * Information about the shipping status update.
     *
     * @var string
     */
    public $update;
}
```

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 array
 */
public function broadcastOn()
{
    return new PrivateChannel('order.'.$this->update->order_id);
}
```

Authorizing Channels

Remember, users must be authorized to listen on private channels. We may define our channel authorization rules in the `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:

```
Broadcast::channel('order.{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 `ShippingStatusUpdated` event. By default, all of the event's public properties will be included on the broadcast event:

```
Echo.private(`order.${orderId}`)
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.update);
    });
```

Defining Broadcast Events

To inform Laravel that a given event should be broadcast, 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 Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

    /**
     * Create a new event instance.
     *
```

```
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

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

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

Broadcast 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';
}
```

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 used by the broadcaster by defining a `broadcastQueue` property on your event class. This property should specify the name of the queue you wish to use when broadcasting:

```
/**
 * The name of the queue on which to place the event.
 *
 * @var string
 */
public $broadcastQueue = 'your-queue-name';
```

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 `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);
```

Defining Authorization Callbacks

Next, we need to define the logic that will actually perform the channel authorization. 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('order.{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 the string or numeric order ID, you may request an actual `Order` model instance:

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

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` function. The event dispatcher will notice that the event is marked with the `ShouldBroadcast` interface and will queue the event for broadcasting:

```
event(new ShippingStatusUpdated($update));
```

Only To Others

When building an application that utilizes event broadcasting, you may substitute the `event` function with the `broadcast` function. Like the `event` function, the `broadcast` function dispatches the event to your server-side listeners:

```
broadcast(new ShippingStatusUpdated($update));
```

However, the `broadcast` function also exposes the `toOthers` method which allows you to exclude the current user from the broadcast's recipients:

```
broadcast(new ShippingStatusUpdated($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` end-point 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 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.

Configuration

When you initialize a Laravel Echo instance, a socket ID is assigned to the connection. If you are using Vue and Axios, 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 Vue and Axios, you will need to manually configure your JavaScript application to send the `X-Socket-ID` header. You may retrieve the socket ID using the `Echo.socketId` method:

```
var socketId = Echo.socketId();
```

Receiving Broadcasts

Installing Laravel Echo

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by Laravel. 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 broadcaster:

```
npm install --save 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/assets/js/bootstrap.js` file that is included with the Laravel framework:

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

When creating an Echo instance that uses the `pusher` connector, you may also specify a `cluster` as well as whether the connection should be encrypted:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key',
    cluster: 'eu',
    encrypted: true
});
```

Listening For Events

Once you have installed and instantiated Echo, you are ready to start listening for event broadcasts. 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')
    .listen('OrderShipped', (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')
    .listen(...)
    .listen(...)
    .listen(...);
```

Leaving A Channel

To leave a channel, you may call the `leave` method on your Echo instance:

```
Echo.leave('orders');
```

Namespaces

You may have noticed in the examples above that we did not specify the full 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',
    key: 'your-pusher-key',
    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.

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.*', 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);
}
```

Like public or private events, presence channel events may be broadcast using the `broadcast` function. As with other events, you may use the `toOthers` method to exclude the current user from receiving the broadcast:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

You may listen for the join event via Echo's `listen` method:

```
Echo.join(`chat.${roomId}`)
    .here(...)
    .joining(...)
    .leaving(...)
    .listen('NewMessage', (e) => {
        //
    });
```

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.channel('chat')
    .whisper('typing', {
        name: this.user.name
    });
```

To listen for client events, you may use the `listenForWhisper` method:

```
Echo.channel('chat')
    .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. First, 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.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
    });
```

In this example, all notifications sent to `App\User` instances via the `broadcast` channel would be received by the callback. A channel authorization callback for the `App.User.{id}` channel is included in the default `BroadcastServiceProvider` that ships with the Laravel framework.

Cache

Configuration

Laravel provides an expressive, unified API for various caching backends. The cache configuration is located at `config/cache.php`. In this file you may specify which cache driver you would like used by default throughout your application. Laravel supports popular caching backends like Memcached and Redis out of the box.

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 in the 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:

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
```

```
],
```

You may also 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 `predis/predis` package (~1.0) via Composer or install the PhpRedis PHP extension via PECL.

For more information on configuring Redis, consult its Laravel documentation page.

Cache Usage

Obtaining A Cache Instance

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

However, you may also use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the 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', 10);
```

Retrieving Items From The Cache

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

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

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

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. When you place an item in the cache, you need to specify the number of minutes for which the value should be cached:

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

Instead of passing the number of minutes as an integer, you may also pass a `DateTime` instance representing the expiration time of the cached item:

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

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

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`:

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

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 clear the entire cache using the `flush` method:

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

> {note} Flushing the cache does not respect the 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 or cache contract, 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'], $minutes);

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

{tip} When testing call to the global `cache` function, you may use the `Cache::shouldReceive` method just as if you were testing a facade.

Cache Tags

{note} Cache tags are not supported when using the `file` 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` value in the cache:

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

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 caches tagged with `authors`, so `Anne` would be removed, but not `John`:

```
Cache::tags('authors')->flush();
```

Adding Custom Cache Drivers

Writing The Driver

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` contract contract. So, a MongoDB cache implementation would 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, $minutes) {}
    public function putMany(array $values, $minutes);
    public function increment($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 framework source code. Once our implementation is complete, we can finish our custom driver registration.

```
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. The call to `Cache::extend` could be done in the `boot` method of the default `App\Providers\AppServiceProvider` that ships with fresh Laravel applications, or you may create your own service provider to house the extension - just don't forget to register the provider in the `config/app.php` provider array:

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function ($app) {
            return Cache::repository(new MongoStore);
        });
    }

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

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, simply 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 `EventServiceProvider`:

```
/**
 * 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',
    ],
];
```

Collections

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.

Available Methods

For the remainder of this documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulating 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 collapse combine contains containsStrict count diff diffKeys each every except filter first flatMap flatten flip forget forPage get groupBy has implode intersect isEmpty isNotEmpty keyBy keys last map mapWithKeys max median merge min mode nth only partition pipe pluck pop prepend pull push put random reduce reject reverse search shift shuffle slice sort sortBy sortByDesc splice split sum take tap times toArray toJson transform union unique uniqueStrict values when where whereStrict whereIn whereInStrict whereNotIn whereNotInStrict zip

Method Listing

`all()` {#collection-method .first-collection-method}

The `all` method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

`average()` {#collection-method}

Alias for the `avg` method.

`avg()` {#collection-method}

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()` {#collection-method}

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->toArray();

// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. Imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
```

```
    </div>
@endforeach
```

`collapse()` {#collection-method}

The `collapse` method collapses a collection of arrays into a single, flat collection:

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`combine()` {#collection-method}

The `combine` method combines the keys of the collection with the values of another array or collection:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

`contains()` {#collection-method}

The `contains` method determines whether the collection contains a given item:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

You may also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
```

```
]);

$collection->contains('product', 'Bookcase');

// false
```

Finally, you may also pass a callback to the `contains` method to perform your own truth test:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($value, $key) {
    return $value > 5;
});

// false
```

The `contains` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `containsStrict` method to filter using "strict" comparisons.

`containsStrict()` {#collection-method}

This method has the same signature as the `contains` method; however, all values are compared using "strict" comparisons.

`count()` {#collection-method}

The `count` method returns the total number of items in the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

`diff()` {#collection-method}

The `diff` method compares the collection against another collection or a plain PHP `array` based on its values. This method will return the values in the original collection that are not present in the given collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();
```

```
// [1, 3, 5]
```

`diffKeys()` {#collection-method}

The `diffKeys` method compares the collection against another collection or a plain PHP `array` based on its keys. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

`each()` {#collection-method}

The `each` method iterates over the items in the collection and passes each item to a callback:

```
$collection = $collection->each(function ($item, $key) {
    //
});
```

If you would like to stop iterating through the items, you may return `false` from your callback:

```
$collection = $collection->each(function ($item, $key) {
    if (/* some condition */) {
        return false;
    }
});
```

every() {#collection-method}

The every method may be used to verify that all elements of a collection pass a given truth test:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

except() {#collection-method}

The except method returns all items in the collection except for those with the specified keys:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

For the inverse of except, see the only method.

filter() {#collection-method}

The filter method filters the collection using the given callback, keeping only those items that pass a given truth test:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to false will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

For the inverse of `filter`, see the reject method.

`first()` {#collection-method}

The `first` method returns the first element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
    return $value > 2;
});

// 3
```

You may also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->first();

// 1
```

`flatMap()` {#collection-method}

The `flatMap` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items. Then, the array is flattened by a level:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

`flatten()` {#collection-method}

The `flatten` method flattens a multi-dimensional collection into a single dimension:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);
```

```
$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

You may optionally pass the function a "depth" argument:

```
$collection = collect([
    'Apple' => [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ],
    'Samsung' => [
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']
    ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
    [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
        ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
    ]
*/
```

In this example, calling `flatten` without providing the depth would have also flattened the nested arrays, resulting in `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Providing a depth allows you to restrict the levels of nested arrays that will be flattened.

`flip()` {#collection-method}

The `flip` method swaps the collection's keys with their corresponding values:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

`forget()` {#collection-method}

The `forget` method removes an item from the collection by its key:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

> {note} Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

`forPage()` {#collection-method}

The `forPage` method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

`get()` {#collection-method}

The `get` method returns the item at a given key. If the key does not exist, `null` is returned:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

You may optionally pass a default value as the second argument:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

You may even pass a callback as the default value. The result of the callback will be returned if the specified key does not exist:

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

groupBy() {#collection-method}

The groupBy method groups the collection's items by a given key:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->toArray();

/*
    [
        'account-x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'account-x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

In addition to passing a string key, you may also pass a callback. The callback should return the value you wish to key the group by:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->toArray();

/*
    [
        'x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
```

```
        'x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

`has()` {#collection-method}

The `has` method determines if a given key exists in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);

$collection->has('product');

// true
```

`implode()` {#collection-method}

The `implode` method joins the items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

If the collection contains simple strings or numeric values, simply pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

`intersect()` {#collection-method}

The `intersect` method removes any values from the original collection that are not present in the given `array` or collection. The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);
```

```
$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

isEmpty() {#collection-method}

The `isEmpty` method returns `true` if the collection is empty; otherwise, `false` is returned:

```
collect([])->isEmpty();

// true
```

isNotEmpty() {#collection-method}

The `isNotEmpty` method returns `true` if the collection is not empty; otherwise, `false` is returned:

```
collect([])->isNotEmpty();

// false
```

keyBy() {#collection-method}

The `keyBy` method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'desk'],
    ['product_id' => 'prod-200', 'name' => 'chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

`keys()` {#collection-method}

The `keys` method returns all of the collection's keys:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

`last()` {#collection-method}

The `last` method returns the last element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});

// 2
```

You may also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->last();

// 4
```

`map()` {#collection-method}

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

> {note} Like most other collection methods, `map` returns a new collection instance; it does not modify the collection it is called on. If you want to transform the original collection, use the `transform` method.

`mapWithKeys()` {#collection-method}

The `mapWithKeys` method iterates through the collection and passes each value to the given callback. The callback should return an associative array containing a single key / value pair:

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com'
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com'
    ]
]);

$keyed = $collection->mapWithKeys(function ($item) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
    [
```

```
        'john@example.com' => 'John',
        'jane@example.com' => 'Jane',
    ]
*/
```

`max()` {#collection-method}

The `max` method returns the maximum value of a given key:

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

`median()` {#collection-method}

The `median` method returns the median value of a given key:

```
$median = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->median('foo

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

`merge()` {#collection-method}

The `merge` method merges the given array or collection with the original collection. If a string key in the given items matches a string key in the original collection, the given items's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given items's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);
```

```
$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min() {#collection-method}

The `min` method returns the minimum value of a given key:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

mode() {#collection-method}

The `mode` method returns the mode value of a given key:

```
$mode = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]
```

nth() {#collection-method}

The `nth` method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

You may optionally pass an offset as the second argument:

```
$collection->nth(4, 1);

// ['b', 'f']
```

only() {#collection-method}

The only method returns the items in the collection with the specified keys:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => fa

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

For the inverse of only, see the except method.

partition() {#collection-method}

The partition method may be combined with the list PHP function to separate elements that pass a given truth test from those that do not:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $aboveThree) = $collection->partition(function ($i) {
    return $i < 3;
});
```

pipe() {#collection-method}

The pipe method passes the collection to the given callback and returns the result:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

pluck() {#collection-method}

The pluck method retrieves all of the values for a given key:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');
```

```
$plucked->all();

// ['Desk', 'Chair']
```

You may also specify how you wish the resulting collection to be keyed:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop() {#collection-method}

The pop method removes and returns the last item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

prepend() {#collection-method}

The prepend method adds an item to the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to set the key of the prepended item:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

pull() {#collection-method}

The pull method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

push() {#collection-method}

The push method appends an item to the end of the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

put() {#collection-method}

The put method sets the given key and value in the collection:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random() {#collection-method}

The random method returns a random item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

You may optionally pass an integer to `random` to specify how many items you would like to randomly retrieve. A collection of items is always returned when explicitly passing the number of items you wish to receive:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

`reduce()` {#collection-method}

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

The value for `$carry` on the first iteration is `null`; however, you may specify its initial value by passing a second argument to `reduce`:

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

`reject()` {#collection-method}

The `reject` method filters the collection using the given callback. The callback should return `true` if the item should be removed from the resulting collection:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

For the inverse of the `reject` method, see the `filter` method.

`reverse()` {#collection-method}

The `reverse` method reverses the order of the collection's items:

```
$collection = collect([1, 2, 3, 4, 5]);

$reversed = $collection->reverse();

$reversed->all();

// [5, 4, 3, 2, 1]
```

`search()` {#collection-method}

The `search` method searches the collection for the given value and returns its key if found. If the item is not found, `false` is returned.

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value. To use "strict" comparison, pass `true` as the second argument to the method:

```
$collection->search('4', true);

// false
```

Alternatively, you may pass in your own callback to search for the first item that passes your truth test:

```
$collection->search(function ($item, $key) {
    return $item > 5;
});

// 2
```

`shift()` {#collection-method}

The `shift` method removes and returns the first item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1
```

```
$collection->all();

// [2, 3, 4, 5]
```

`shuffle()` {#collection-method}

The `shuffle` method randomly shuffles the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

`slice()` {#collection-method}

The `slice` method returns a slice of the collection starting at the given index:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

The returned slice will preserve keys by default. If you do not wish to preserve the original keys, you can use the `values` method to reindex them.

`sort()` {#collection-method}

The `sort` method sorts the collection. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

If your sorting needs are more advanced, you may pass a callback to `sort` with your own algorithm. Refer to the PHP documentation on `usort`, which is what the collection's `sort` method calls under the hood.

> {tip} If you need to sort a collection of nested arrays or objects, see the `sortBy` and `sortByDesc` methods.

`sortBy()` {#collection-method}

The `sortBy` method sorts the collection by the given key. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

You can also pass your own callback to determine how to sort the collection values:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
```

```
$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
    ]
*/
```

`sortByDesc()` {#collection-method}

This method has the same signature as the `sortBy` method, but will sort the collection in the opposite order.

`splice()` {#collection-method}

The `splice` method removes and returns a slice of items starting at the specified index:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

You may pass a second argument to limit the size of the resulting chunk:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();
```

```
// [1, 2, 4, 5]
```

In addition, you can pass a third argument containing the new items to replace the items removed from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

`split()` {#collection-method}

The `split` method breaks a collection into the given number of groups:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->toArray();

// [[1, 2], [3, 4], [5]]
```

`sum()` {#collection-method}

The `sum` method returns the sum of all items in the collection:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

If the collection contains nested arrays or objects, you should pass a key to use for determining which values to sum:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');
```

```
// 1272
```

In addition, you may pass your own callback to determine which values of the collection to sum:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});

// 6
```

`take()` {#collection-method}

The `take` method returns a new collection with the specified number of items:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

You may also pass a negative integer to take the specified amount of items from the end of the collection:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

`tap()` {#collection-method}

The `tap` method passes the collection to the given callback, allowing you to "tap" into the collection at a specific point and do something with the items while not affecting the collection itself:

```
collect([2, 4, 3, 1, 5])
    ->sort()
```

```
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->toArray());
    })
    ->shift();

// 1
```

`times()` {#collection-method}

The static `times` method creates a new collection by invoking the callback a given amount of times:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

This method can be useful when combined with factories to create Eloquent models:

```
$categories = Collection::times(3, function ($number) {
    return factory(Category::class)->create(['name' => 'Category #'.$number]);
});

$categories->all();

/*
    [
        ['id' => 1, 'name' => 'Category #1'],
        ['id' => 2, 'name' => 'Category #2'],
        ['id' => 3, 'name' => 'Category #3'],
    ]
*/
```

`toArray()` {#collection-method}

The `toArray` method converts the collection into a plain PHP `array`. If the collection's values are Eloquent models, the models will also be converted to arrays:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();
```

```
/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

{note} `toArray` also converts all of the collection's nested objects to an array. If you want to get the raw underlying array, use the `all` method instead.

`toJson()` {#collection-method}

The `toJson` method converts the collection into a JSON serialized string:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

`transform()` {#collection-method}

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

{note} Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the `map` method.

`union()` {#collection-method}

The `union` method adds the given array to the collection. If the given array contains keys that are already in the original collection, the original collection's values will be preferred:

```
$collection = collect([1 => ['a'], 2 => ['b']]);
```

```
$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

`unique()` {#collection-method}

The `unique` method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

When dealing with nested arrays or objects, you may specify the key used to determine uniqueness:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ]
*/
```

You may also pass your own callback to determine item uniqueness:

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});
```

```
$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
    ]
*/
```

The `unique` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `uniqueStrict` method to filter using "strict" comparisons.

`uniqueStrict()` {#collection-method}

This method has the same signature as the `unique` method; however, all values are compared using "strict" comparisons.

`values()` {#collection-method}

The `values` method returns a new collection with the keys reset to consecutive integers:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);

$values = $collection->values();

$values->all();

/*
    [
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

`when()` {#collection-method}

The `when` method will execute the given callback when the first argument given to the method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});

$collection->all();

// [1, 2, 3, 4]
```

`where()` {#collection-method}

The `where` method filters the collection by a given key / value pair:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `where` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereStrict` method to filter using "strict" comparisons.

`whereStrict()` {#collection-method}

This method has the same signature as the `where` method; however, all values are compared using "strict" comparisons.

`whereIn()` {#collection-method}

The `whereIn` method filters the collection by a given key / value contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Bookcase', 'price' => 150],
        ['product' => 'Desk', 'price' => 200],
    ]
*/
```

The `whereIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereInStrict` method to filter using "strict" comparisons.

`whereInStrict()` {#collection-method}

This method has the same signature as the `whereIn` method; however, all values are compared using "strict" comparisons.

`whereNotIn()` {#collection-method}

The `whereNotIn` method filters the collection by a given key / value not contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
    [
```

```
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `whereNotIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereNotInStrict` method to filter using "strict" comparisons.

`whereNotInStrict()` {#collection-method}

This method has the same signature as the `whereNotIn` method; however, all values are compared using "strict" comparisons.

`zip()` {#collection-method}

The `zip` method merges together the values of the given array with the values of the original collection at the corresponding index:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Higher Order Messages

Collections also provide support for "higher order messages", which are shortcuts for performing common actions on collections. The collection methods that provide higher order messages are: `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `map`, `partition`, `reject`, `sortBy`, `sortByDesc`, and `sum`.

Each higher order message can be accessed as a dynamic property on a collection instance. For instance, let's use the `each` higher order message to call a method on each object within a collection:

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Likewise, we can use the `sum` higher order message to gather the total number of "votes" for a collection of users:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Events

Introduction

Laravel's events provides a simple observer implementation, allowing you to subscribe and listen for various events that occur in 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, since 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 simply raise an `OrderShipped` event, which a listener can receive and transform into a Slack notification.

Registering Events & Listeners

The `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). Of course, you may add as many events to this array as your application requires. For example, let's add a `OrderShipped` event:

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

Generating Events & Listeners

Of course, manually creating the files for each event and listener is cumbersome. Instead, simply add listeners and events to your `EventServiceProvider` and use the `event:generate` command. This command will generate any events or listeners that are listed in your `EventServiceProvider`. Of course, events and listeners that already exist will be left untouched:

```
php artisan event:generate
```

Manually Registering Events

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register Closure based events manually in the `boot` method of your `EventServiceProvider`:

```
/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    parent::boot();

    Event::listen('event.name', function ($foo, $bar) {
        //
    });
}
```

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) {
    //
});
```

Defining Events

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

```
<?php

namespace App\Events;

use App\Order;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use SerializesModels;

    public $order;

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

As you can see, this event class contains no logic. It is simply a container for the `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.

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive the event instance in their `handle` method. The `event:generate` command 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  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 e-mail or making an HTTP request. Before getting started with queued listeners, make sure to configure your queue and start a queue listener 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` Artisan command 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 this listener is called for an event, it will be automatically 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 and queue name used by an event listener, you may define `$connection` and `$queue` 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';
}
```

Manually Accessing 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\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

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

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 exception that caused the failure:

```
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(OrderShipped $event)
    {
        //
    }

    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

Dispatching Events

To dispatch an event, you may pass an instance of the event to the `event` helper. The helper will dispatch the event to all of its registered listeners. Since the `event` helper is globally available, you may call it from anywhere in your application:

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
```

```
     * @param  int  $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // Order shipment logic...

        event(new OrderShipped($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 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 onUserLogin($event) {}

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

    /**
     * Register the listeners for the subscriber.
     *
```

```
     * @param  Illuminate\Events\Dispatcher  $events
     */
    public function subscribe($events)
    {
        $events->listen(
            'Illuminate\Auth\Events\Login',
            'App\Listeners\UserEventSubscriber@onUserLogin'
        );

        $events->listen(
            'Illuminate\Auth\Events\Logout',
            'App\Listeners\UserEventSubscriber@onUserLogout'
        );
    }

}
```

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

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

    /**
     * The subscriber classes to register.
     *
     * @var array
```

```
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}
```

File Storage

Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple to use drivers for working with local filesystems, Amazon S3, and Rackspace Cloud Storage. Even better, it's amazingly simple to switch between these storage options as the API remains the same for each system.

Configuration

The filesystem configuration file is located at `config/filesystems.php`. Within this file you may configure all of your "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file. So, simply modify the configuration to reflect your storage preferences and credentials.

Of course, you may configure as many disks as you like, and may even have multiple disks that use the same driver.

The Public Disk

The `public` disk is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores these files in `storage/app/public`. To make them accessible from the web, you should create a symbolic link from `public/storage` to `storage/app/public`. This 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
```

Of course, 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');
```

The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory defined in your configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would store a file in `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Driver Prerequisites

Composer Packages

Before using the S3 or Rackspace drivers, you will need to install the appropriate package via Composer:

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~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.

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 example configuration 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,
],
```

Rackspace Driver Configuration

Laravel's Flysystem integrations works great with Rackspace; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure a Rackspace filesystem, you may use the example configuration below:

```
'rackspace' => [
    'driver'    => 'rackspace',
    'username'  => 'your-username',
    'key'       => 'your-key',
    'container' => 'your-container',
    'endpoint'  => 'https://identity.api.rackspacecloud.com/v2.0/',
    'region'    => 'IAD',
    'url_type'  => 'publicURL',
],
```

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 call will automatically be passed to the default disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $fileContents);
```

If your applications 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', $fileContents);
```

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 "root" location configured for the disk:

```
$contents = Storage::get('file.jpg');
```

The `exists` method may be used to determine if a file exists on the disk:

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

File URLs

When using the `local` or `s3` drivers, you may use the `url` method to get the URL for the 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('file1.jpg');
```

> {note} Remember, if you are 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.

Local URL Host Customization

If you would like to pre-define the host for files stored on a disk using the `local` driver, 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 the file in bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file1.jpg');
```

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file1.jpg');
```

Storing Files

The `put` method may be used to store raw file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Using streams is greatly recommended when dealing with large files:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Automatic Streaming

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 a `Illuminate\Http\File` or `Illuminate\Http\UploadedFile` instance and will automatically stream the file to your desired location:

```
use Illuminate\Http\File;

// Automatically generate a unique ID for file name...
Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a file name...
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, not a file name. By default, the `putFile`

method will generate a unique ID to serve as the file name. The path to the file will be returned by the `putFile` method so you can store the path, including the generated file name, 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 S3 and would like the file to be publicly accessible:

```
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/file1.jpg', 'new/file1.jpg');

Storage::move('old/file1.jpg', 'new/file1.jpg');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as profile pictures, photos, and documents. Laravel makes it very easy to store uploaded files using the `store` method on an uploaded file instance. Simply call the `store` method with the path at which you wish to store the uploaded file:

```
<?php

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
```

```
     * Update the avatar for the user.
     *
     * @param  Request  $request
     * @return 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 file name. By default, the `store` method will generate a unique ID to serve as the file name. The path to the file will be returned by the `store` method so you can store the path, including the generated file name, in your database.

You may also call the `putFile` method on the `Storage` facade to perform the same file manipulation as the example above:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying A File Name

If you would not like a file name to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the file name, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

Of course, you may also use the `putFileAs` method on the `Storage` facade, which will perform the same file manipulation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

Specifying A Disk

By default, this 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'
);
```

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 setting 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')
```

Deleting Files

The `delete` method accepts a single filename or an array of files to remove from the disk:

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.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 sub-directories, 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 sub-directories:

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

Create A Directory

The `makeDirectory` method will create the given directory, including any needed sub-directories:

```
Storage::makeDirectory($directory);
```

Delete A Directory

Finally, the `deleteDirectory` may be used to remove a directory and all of its files:

```
Storage::deleteDirectory($directory);
```

Custom Filesystems

Laravel's Flysystem integration provides drivers 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 set up the 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 should create a service provider such as `DropboxServiceProvider`. In the provider's `boot` method, you may use the `Storage` facade's `extend` method to define the custom driver:

```
<?php

namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
```

```
use Illuminate\Support\ServiceProvider;
use Spatie\FlysystemDropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorizationToken']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

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

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 resolver 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 the service provider to register the extension, you may use the `dropbox` driver in your `config/filesystems.php` configuration file.

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

array_add array_collapse array_divide array_dot array_except array_first array_flatten array_forget array_get array_has array_last array_only array_pluck array_prepend array_pull array_set array_sort array_sort_recursive array_where array_wrap head last

Paths

app_path base_path config_path database_path mix public_path resource_path storage_path

Strings

camel_case class_basename e ends_with kebab_case snake_case str_limit starts_with str_contains str_finish str_is str_plural str_random str_singular str_slug studly_case title_case trans trans_choice

URLs

action asset secure_asset route secure_url url

Miscellaneous

abort abort_if abort_unless auth back bcrypt cache collect config csrf_field csrf_token dd dispatch env event factory info logger method_field old redirect request response retry session value view

Method Listing

Arrays

`array_add()` {#collection-method .first-collection-method}

The `array_add` function adds a given key / value pair to the array if the given key doesn't already exist in the array:

```
$array = array_add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

`array_collapse()` {#collection-method}

The `array_collapse` function collapses an array of arrays into a single array:

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`array_divide()` {#collection-method}

The `array_divide` function returns two arrays, one containing the keys, and the other containing the values of the original array:

```
list($keys, $values) = array_divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

`array_dot()` {#collection-method}

The `array_dot` function flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth:

```
$array = array_dot(['foo' => ['bar' => 'baz']]);

// ['foo.bar' => 'baz'];
```

`array_except()` {#collection-method}

The `array_except` function removes the given key / value pairs from the array:

```
$array = ['name' => 'Desk', 'price' => 100];

$array = array_except($array, ['price']);

// ['name' => 'Desk']
```

`array_first()` {#collection-method}

The `array_first` function returns the first element of an array passing a given truth test:

```
$array = [100, 200, 300];

$value = array_first($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:

```
$value = array_first($array, $callback, $default);
```

`array_flatten()` {#collection-method}

The `array_flatten` function will flatten a multi-dimensional array into a single level.

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$array = array_flatten($array);

// ['Joe', 'PHP', 'Ruby'];
```

`array_forget()` {#collection-method}

The `array_forget` function removes a given key / value pair from a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

array_forget($array, 'products.desk');

// ['products' => []]
```

`array_get()` {#collection-method}

The `array_get` function retrieves a value from a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

$value = array_get($array, 'products.desk');
```

```
// ['price' => 100]
```

The `array_get` function also accepts a default value, which will be returned if the specific key is not found:

```
$value = array_get($array, 'names.john', 'default');
```

`array_has()` {#collection-method}

The `array_has` function checks that a given item or items exists in an array using "dot" notation:

```
$array = ['product' => ['name' => 'desk', 'price' => 100]];

$hasItem = array_has($array, 'product.name');

// true

$hasItems = array_has($array, ['product.price', 'product.discount']);

// false
```

`array_last()` {#collection-method}

The `array_last` function returns the last element of an array passing a given truth test:

```
$array = [100, 200, 300, 110];

$value = array_last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

`array_only()` {#collection-method}

The `array_only` function will return only the specified key / value pairs from the given array:

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$array = array_only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

`array_pluck()` {#collection-method}

The `array_pluck` function will pluck a list of the given key / value pairs from the array:

```
$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$array = array_pluck($array, 'developer.name');

// ['Taylor', 'Abigail'];
```

You may also specify how you wish the resulting list to be keyed:

```
$array = array_pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail'];
```

`array_prepend()` {#collection-method}

The `array_prepend` function will push an item onto the beginning of an array:

```
$array = ['one', 'two', 'three', 'four'];

$array = array_prepend($array, 'zero');

// $array: ['zero', 'one', 'two', 'three', 'four']
```

`array_pull()` {#collection-method}

The `array_pull` function returns and removes a key / value pair from the array:

```
$array = ['name' => 'Desk', 'price' => 100];

$name = array_pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

`array_set()` {#collection-method}

The `array_set` function sets a value within a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];
```

```
array_set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

`array_sort()` {#collection-method}

The `array_sort` function sorts the array by the results of the given Closure:

```
$array = [
    ['name' => 'Desk'],
    ['name' => 'Chair'],
];

$array = array_values(array_sort($array, function ($value) {
    return $value['name'];
}));

/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
    ]
*/
```

`array_sort_recursive()` {#collection-method}

The `array_sort_recursive` function recursively sorts the array using the `sort` function:

```
$array = [
    [
        'Roman',
        'Taylor',
        'Li',
    ],
    [
        'PHP',
        'Ruby',
        'JavaScript',
    ],
];

$array = array_sort_recursive($array);

/*
    [
```

```
        [
            'Li',
            'Roman',
            'Taylor',
        ],
        [
            'JavaScript',
            'PHP',
            'Ruby',
        ]
    ];
*/
```

`array_where()` {#collection-method}

The `array_where` function filters the array using the given Closure:

```
$array = [100, '200', 300, '400', 500];

$array = array_where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => 200, 3 => 400]
```

`array_wrap()` {#collection-method}

The `array_wrap` function will wrap the given value in an array. If the given value is already an array it will not be changed:

```
$string = 'Laravel';

$array = array_wrap($string);

// [0 => 'Laravel']
```

`head()` {#collection-method}

The `head` function returns the first element in the given array:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

`last()` {#collection-method}

The `last` function returns the last element in the given array:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

Paths

`app_path()` {#collection-method}

The `app_path` function returns the fully qualified path to the `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()` {#collection-method}

The `base_path` function returns the fully qualified path to the project root. 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()` {#collection-method}

The `config_path` function returns the fully qualified path to the application configuration directory:

```
$path = config_path();
```

`database_path()` {#collection-method}

The `database_path` function returns the fully qualified path to the application's database directory:

```
$path = database_path();
```

`mix()` {#collection-method}

The `mix` function gets the path to a versioned Mix file:

```
mix($file);
```

`public_path()` {#collection-method}

The `public_path` function returns the fully qualified path to the `public` directory:

```
$path = public_path();
```

`resource_path()` {#collection-method}

The `resource_path` function returns the fully qualified path to the `resources` directory. You may also use the `resource_path` function to generate a fully qualified path to a given file relative to the resources directory:

```
$path = resource_path();

$path = resource_path('assets/sass/app.scss');
```

`storage_path()` {#collection-method}

The `storage_path` function returns the fully qualified path to the `storage` directory. You may also use the `storage_path` function to generate a fully qualified path to a given file relative to the storage directory:

```
$path = storage_path();

$path = storage_path('app/file.txt');
```

Strings

`camel_case()` {#collection-method}

The `camel_case` function converts the given string to `camelCase`:

```
$camel = camel_case('foo_bar');

// fooBar
```

`class_basename()` {#collection-method}

The `class_basename` returns the class name of the given class with the class' namespace removed:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e() {#collection-method}

The e function runs PHP's htmlspecialchars function with the double_encode option set to false:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

ends_with() {#collection-method}

The ends_with function determines if the given string ends with the given value:

```
$value = ends_with('This is my name', 'name');

// true
```

kebab_case() {#collection-method}

The kebab_case function converts the given string to kebab-case:

```
$value = kebab_case('fooBar');

// foo-bar
```

snake_case() {#collection-method}

The snake_case function converts the given string to snake_case:

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit() {#collection-method}

The str_limit function limits the number of characters in a string. The function accepts a string as its first argument and the maximum number of resulting characters as its second argument:

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

`starts_with()` {#collection-method}

The `starts_with` function determines if the given string begins with the given value:

```
$value = starts_with('This is my name', 'This');

// true
```

`str_contains()` {#collection-method}

The `str_contains` function determines if the given string contains the given value:

```
$value = 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:

```
$value = str_contains('This is my name', ['my', 'foo']);

// true
```

`str_finish()` {#collection-method}

The `str_finish` function adds a single instance of the given value to a string:

```
$string = str_finish('this/string', '/');

// this/string/
```

`str_is()` {#collection-method}

The `str_is` function determines if a given string matches a given pattern. Asterisks may be used to indicate wildcards:

```
$value = str_is('foo*', 'foobar');

// true

$value = str_is('baz*', 'foobar');

// false
```

str_plural() {#collection-method}

The str_plural function converts a string to its plural form. This function currently only supports the English language:

```
$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:

```
$plural = str_plural('child', 2);

// children

$plural = str_plural('child', 1);

// child
```

str_random() {#collection-method}

The str_random function generates a random string of the specified length. This function uses PHP's random_bytes function:

```
$string = str_random(40);
```

str_singular() {#collection-method}

The str_singular function converts a string to its singular form. This function currently only supports the English language:

```
$singular = str_singular('cars');

// car
```

str_slug() {#collection-method}

The str_slug function generates a URL friendly "slug" from the given string:

```
$title = str_slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

`studly_case()` {#collection-method}

The `studly_case` function converts the given string to `StudlyCase`:

```
$value = studly_case('foo_bar');

// FooBar
```

`title_case()` {#collection-method}

The `title_case` function converts the given string to `Title Case`:

```
$title = title_case('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

`trans()` {#collection-method}

The `trans` function translates the given language line using your localization files:

```
echo trans('validation.required'):
```

`trans_choice()` {#collection-method}

The `trans_choice` function translates the given language line with inflection:

```
$value = trans_choice('foo.bar', $count);
```

URLs

`action()` {#collection-method}

The `action` function generates a URL for the given controller action. You do not need to pass the full namespace to the controller. Instead, pass the controller class name relative to the `App\Http\Controllers` namespace:

```
$url = action('HomeController@getIndex');
```

If the method accepts route parameters, you may pass them as the second argument to the method:

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

`asset()` {#collection-method}

Generate a URL for an asset using the current scheme of the request (HTTP or HTTPS):

```
$url = asset('img/photo.jpg');
```

`secure_asset()` {#collection-method}

Generate a URL for an asset using HTTPS:

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

`route()` {#collection-method}

The `route` function generates a URL for the given named route:

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

If the route accepts parameters, you may pass them as the second argument to the method:

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

`secure_url()` {#collection-method}

The `secure_url` function generates a fully qualified HTTPS URL to the given path:

```
echo secure_url('user/profile');

echo secure_url('user/profile', [1]);
```

`url()` {#collection-method}

The `url` function generates a fully qualified URL to the given path:

```
echo url('user/profile');

echo url('user/profile', [1]);
```

If no path is provided, a `Illuminate\Routing\UrlGenerator` instance is returned:

```
echo url()->current();
echo url()->full();
echo url()->previous();
```

Miscellaneous

`abort()` {#collection-method}

The `abort` function throws a HTTP exception which will be rendered by the exception handler:

```
abort(401);
```

You may also provide the exception's response text:

```
abort(401, 'Unauthorized.');
```

`abort_if()` {#collection-method}

The `abort_if` function throws an HTTP exception if a given boolean expression evaluates to `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

`abort_unless()` {#collection-method}

The `abort_unless` function throws an HTTP exception if a given boolean expression evaluates to `false`:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

`auth()` {#collection-method}

The `auth` function returns an authenticator instance. You may use it instead of the `Auth` facade for convenience:

```
$user = auth()->user();
```

`back()` {#collection-method}

The `back()` function generates a redirect response to the user's previous location:

```
return back();
```

`bcrypt()` {#collection-method}

The `bcrypt` function hashes the given value using Bcrypt. You may use it as an alternative to the `Hash` facade:

```
$password = bcrypt('my-secret-password');
```

`cache()` {#collection-method}

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 minutes or duration the cached value should be considered valid:

```
cache(['key' => 'value'], 5);

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

`collect()` {#collection-method}

The `collect` function creates a collection instance from the given array:

```
$collection = collect(['taylor', 'abigail']);
```

`config()` {#collection-method}

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);
```

The `config` helper may also be used to set configuration variables at runtime by passing an array of key / value pairs:

```
config(['app.debug' => true]);
```

`csrf_field()` {#collection-method}

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()` {#collection-method}

The `csrf_token` function retrieves the value of the current CSRF token:

```
$token = csrf_token();
```

`dd()` {#collection-method}

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:

```
dump($value);
```

`dispatch()` {#collection-method}

The `dispatch` function pushes a new job onto the Laravel job queue:

```
dispatch(new App\Jobs\SendEmails);
```

`env()` {#collection-method}

The `env` function gets the value of an environment variable or returns a default value:

```
$env = env('APP_ENV');

// Return a default value if the variable doesn't exist...
$env = env('APP_ENV', 'production');
```

`event()` {#collection-method}

The `event` function dispatches the given event to its listeners:

```
event(new UserRegistered($user));
```

`factory()` {#collection-method}

The `factory` function creates a model factory builder for a given class, name, and amount. It can be used while testing or seeding:

```
$user = factory(App\User::class)->make();
```

`info()` {#collection-method}

The `info` function will write information to the 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()` {#collection-method}

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()` {#collection-method}

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>
```

`old()` {#collection-method}

The `old` function retrieves an old input value flashed into the session:

```
$value = old('value');

$value = old('value', 'default');
```

`redirect()` {#collection-method}

The `redirect` function returns a redirect HTTP response, or returns the redirector instance if called with no arguments:

```
return redirect('/home');

return redirect()->route('route.name');
```

`request()` {#collection-method}

The `request` function returns the current request instance or obtains an input item:

```
$request = request();

$value = request('key', $default = null)
```

`response()` {#collection-method}

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()` {#collection-method}

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, it's 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()` {#collection-method}

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);
```

`value()` {#collection-method}

The `value` function's behavior will simply return the value it is given. However, if you pass a `Closure` to the function, the `Closure` will be executed then its result will be returned:

```
$value = value(function () {
    return 'bar';
});
```

`view()` {#collection-method}

The `view` function retrieves a view instance:

```
return view('auth.login');
```

Mail

Introduction

Laravel provides a clean, simple API over the popular SwiftMailer library with drivers for SMTP, Mailgun, SparkPost, Amazon SES, PHP's `mail` function, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Driver Prerequisites

The API based drivers such as Mailgun and SparkPost are often simpler and faster than SMTP servers. If possible, you should use one of these drivers. All of the API 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 Guzzle, then set the `driver` 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' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

SparkPost Driver

To use the SparkPost driver, first install Guzzle, then set the `driver` option in your `config/mail.php` configuration file to `sparkpost`. Next, verify that your `config/services.php` configuration file contains the following options:

```
'sparkpost' => [
    'secret' => 'your-sparkpost-key',
],
```

SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library by adding the following line to your `composer.json` file's `require` section and running the `composer update` command:

```
"aws/aws-sdk-php": "~3.0"
```

Next, set the `driver` 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' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region',  // e.g. us-east-1
],
```

Generating Mailables

In Laravel, 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` command:

```
php artisan make:mail OrderShipped
```

Writing Mailables

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'],
```

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 a 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\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
```

```
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var Order
     */
    public $order;

    /**
     * Create a new message instance.
     *
     * @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\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var Order
     */
    protected $order;

    /**
     * Create a new message instance.
     *
     * @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',
                ]);
}
```

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 and retrieving the appropriate CID. 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="{{ $message->embed($pathToFile) }}">
</body>
```

Embedding Raw Data Attachments

If you already have a raw data string you wish to embed into an email template, you may use the `embedData` method on the `$message` variable:

```
<body>
    Here is an image from raw data:

    <img src="{{ $message->embedData($data, $name) }}">
</body>
```

Customizing The SwiftMailer Message

The `withSwiftMessage` method of the `Mailable` base class allows you to register a callback which will be invoked with the raw SwiftMailer message instance before sending the message. This gives you an opportunity to 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', 'HeaderValue');
    });
}
```

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` methods 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');
}
```

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-crafted 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. 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 `blue`, `green`, and `red`. You may add as many button components to a message as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
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 a `html` and a `markdown` 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 mail messages.

> {tip} If you would like to build an entirely new theme for the Markdown components, simply write a new CSS file within the `html/themes` directory and change the `theme` option of your `mail` configuration file.

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 setting the email recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Mail\OrderShipped;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  Request  $request
     * @param  int  $orderId
     * @return Response
     */
    public function ship(Request $request, $orderId)
    {
        $order = Order::findOrFail($orderId);

        // Ship order...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}
```

Of course, you are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients all within a single, chained method call:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

Queueing Mail

Queueing A Mail Message

Since sending email messages can drastically lengthen 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. Of course, 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:

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

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, 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
{
    //
}
```

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. For more information on configuring your application per environment, check out the configuration documentation.

Universal To

Another solution provided by Laravel is to set a universal recipient of all emails sent by the framework. This way, all the emails generated by your application will be sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `to` option in your `config/mail.php` configuration file:

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

Finally, you may use a service like 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.

Events

Laravel fires an event just before sending mail messages. Remember, this event is fired when the mail is *sent*, not when it is queued. You may register an event listener for this event in your `EventServiceProvider`:

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

Notifications

- SMS Notifications
  - Prerequisites
  - Formatting SMS Notifications
  - Customizing The "From" Number
  - Routing SMS Notifications
- Slack Notifications
  - Prerequisites
  - Formatting Slack Notifications
  - Slack Attachments
  - Routing Slack 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 mail, SMS (via Nexmo), and Slack. 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.

Creating Notifications

In Laravel, each notification is represented by a single class (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 optimized 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. First, let's explore us-

ing the trait:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;
}
```

This trait is utilized by the default `App\User` model and contains one method that may be used to send notifications: `notify`. The `notify` method expects to receive a notification instance:

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```

> {tip} Remember, you may use the `Illuminate\Notifications\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 is useful primarily 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:

```
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. Out of the box, 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 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 `make:notification`, so you may immediately add them to your notification class:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

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:

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

```
$user->notify((new InvoicePaid($invoice))->delay($when));
```

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 a `Illuminate\Notifications\Messages\MailMessage` instance. 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 `message` 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 nice, 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` value in your `config/app.php` configuration file. This value will be used in the header and footer of your mail notification messages.

Other 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]
    );
}
```

In addition, you may return a mailable object from the `toMail` method:

```
use App\Mail\InvoicePaid as Mailable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new Mailable($this->invoice))->to($this->user->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 blue:

```
/**
 * 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 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 entity:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @return string
     */
    public function routeNotificationForMail()
    {
        return $this->email_address;
    }
}
```

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 explicit 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 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
```

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:

```
/**
 * 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 `blue`, `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 a `html` and a `markdown` 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.

> {tip} If you would like to build an entirely new theme for the Markdown components, simply write a new CSS file within the `html/themes` directory and change the `theme` option of your `mail` configuration file.

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 custom JSON data 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 client. 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\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:

```
$user = App\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:

```
$user = App\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 URI 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\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\User::find(1);

$user->unreadNotifications()->update(['read_at' => Carbon::now()]);
```

Of course, 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 fired Laravel events from your JavaScript client.

Formatting Broadcast Notifications

The `broadcast` channel broadcasts notifications using Laravel's event broadcasting services, allowing your JavaScript client to catch notifications in realtime. If a notification supports broadcasting, you should define a `toBroadcast` method on the notification class. This method will receive a `$notifiable` entity and should return a `BroadcastMessage` instance. The returned data will be encoded as JSON and broadcast to your JavaScript client. 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 the queue the broadcast operation, you may use the `onConnection` and `onQueue` methods of the `BroadcastMessage`:

```
return (new BroadcastMessage($data))
                ->onConnection('sqs')
                ->onQueue('broadcasts');
```

> {tip} In addition to the data you specify, broadcast notifications will also contain a `type` field containing the class name of the notification.

Listening For Notifications

Notifications will broadcast on a private channel formatted using a `{notifiable}.{id}` convention. So, if you are sending a notification to a `App\User` instance with an ID of `1`, the notification will be broadcast on the `App.User.1` private channel. When using Laravel Echo, you may easily listen for notifications on a channel using the `notification` helper method:

```
Echo.private('App.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
```

Customizing The Notification Channel

If you would like to customize which channels a notifiable entity receives its broadcast notifications on, you may define a `receivesBroadcastNotificationsOn` method on the notifiable entity:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The channels the user receives notification broadcasts on.
```

```
     *
     * @return array
     */
    public function receivesBroadcastNotificationsOn()
    {
        return [
            new PrivateChannel('users.'.$this->id),
        ];
    }
}
```

SMS Notifications

Prerequisites

Sending SMS notifications in Laravel is powered by Nexmo. Before you can send notifications via Nexmo, you need to install the `nexmo/client` Composer package and add a few configuration options to your `config/services.php` configuration file. You may copy the example configuration below to get started:

```
'nexmo' => [
    'key' => env('NEXMO_KEY'),
    'secret' => env('NEXMO_SECRET'),
    '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 Nexmo control panel.

Formatting SMS Notifications

If a notification supports being sent as a SMS, you should define a `toNexmo` method on the notification class. This method will receive a `$notifiable` entity and should return a `Illuminate\Notifications\Messages\NexmoMessage` instance:

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return NexmoMessage
 */
public function toNexmo($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 Nexmo / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your unicode message')
                ->unicode();
}
```

Customizing The "From" Number

If you would like to send some notifications from a phone number that is different from the phone number specified in your `config/services.php` file, you may use the `from` method on a `NexmoMessage` instance:

```
/**
 * Get the Nexmo / 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

When sending notifications via the `nexmo` channel, the notification system will automatically look for a `phone_number` attribute on the notifiable entity. If you

would like to customize the phone number the notification is delivered to, define a `routeNotificationForNexmo` method on the entity:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Nexmo channel.
     *
     * @return string
     */
    public function routeNotificationForNexmo()
    {
        return $this->phone;
    }
}
```

Slack Notifications

Prerequisites

Before you can send notifications via Slack, you must install the Guzzle HTTP library via Composer:

```
composer require guzzlehttp/guzzle
```

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 a `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  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->content('One of your invoices has been paid!');
}
```

In this example we are just sending a single line of text to Slack, which will create a message that looks like the following:

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  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->from('Ghost', ':ghost:')
                ->to('#other')
                ->content('This will be sent to #other');
}
```

You may also use an image as your logo instead of an emoji:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->from('Laravel')
```

```
                ->image('https://laravel.com/favicon.png')
                ->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  $notifiable
 * @return SlackMessage
 */
public function toSlack($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.');
                });
}
```

The example above will generate a Slack message that looks like the following:

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:',
                                ]);
                });
}
```

The example above will create a Slack message that looks like the following:

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:

```
/**
 * 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(['title', 'text']);
                });
}
```

Routing Slack Notifications

To route Slack notifications to the proper location, 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;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @return string
     */
    public function routeNotificationForSlack()
    {
        return $this->slack_webhook_url;
    }
}
```

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  NotificationSent  $event
 * @return void
 */
public function handle(NotificationSent $event)
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

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`:

```
<?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 simply return the class name from the `via` method of any of your notifications:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use App\Channels\VoiceChannel;
use App\Channels\Messages\VoiceMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

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

Package Development

- Introduction
  - A Note On Facades

- Service Providers
- Routing
- Resources
  - Configuration
  - Migrations
  - Routes
  - Translations
  - Views
- 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 an entire BDD testing framework like Behat.

Of course, there are different types of packages. Some packages are stand-alone, meaning they work with any PHP framework. Carbon and Behat are examples of stand-alone packages. Any of these packages may be used with Laravel by simply requesting 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, it is best to use contracts instead of facades. Since your package will not have access to all of Laravel's testing helpers, it will be easier to mock or stub a contract than to mock a facade.

Service Providers

Service providers are the connection points 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.

Routing

To define routes for your package, pass the routes file path to the `loadRoutesFrom` method from within your package service provider's `boot` method. From within your routes file, you may use the `Illuminate\Support\Facades\Route` facade to register routes just as you would within a typical Laravel application:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/path/to/routes.php');
}
```

Resources

Configuration

Typically, you will need to publish your package's configuration file to the application's own `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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/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. Of course, 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. To merge the configurations, use the `mergeConfigFrom` method within your service provider's `register` method:

```
/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__.'/path/to/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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
```

```
    $this->loadRoutesFrom(__DIR__.'/routes.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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/path/to/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 main `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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', '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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => 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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/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('admin', function () {
    return view('courier::admin');
});
```

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` example, Laravel will first check if a custom version of the view has been provided by the developer in `resources/views/vendor/courier`. 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:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/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.

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:

```
/**
 * Bootstrap the application services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            FooCommand::class,
            BarCommand::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 publish groups of related assets:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/assets' => 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 you 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 in the `boot` method of a package service provider:

```
/**
 * Perform post-registration booting of 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
```

Queues

- Introduction
  - Connections Vs. Queues
  - Driver Prerequisites
- Creating Jobs
  - Generating Job Classes
  - Class Structure
- Dispatching Jobs
  - Delayed Dispatching
  - Customizing The Queue & Connection
  - Specifying Max Job Attempts / Timeout Values
  - Error Handling
- Running The Queue Worker
  - Queue Priorities
  - Queue Workers & Deployment
  - Job Expirations & Timeouts
- Supervisor Configuration
- Dealing With Failed Jobs

Introduction

Laravel queues provide a unified API across a variety of different queue backends, such as Beanstalk, Amazon SQS, Redis, or even a relational database. Queues allow you to defer the processing of a time consuming task, such as sending an email, until a later time. Deferring these time consuming tasks drastically speeds up web requests to your application.

The queue configuration file is stored in `config/queue.php`. In this file you will find connection configurations for each of the queue drivers that are included with the framework, which includes a database, Beanstalkd, Amazon SQS, Redis, and a synchronous driver that will execute jobs immediately (for local use). A `null` queue driver is also included which simply discards queued jobs.

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 option. This option defines a particular connection to a backend service 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:

```
// This job is sent to the default queue...
dispatch(new Job);

// This job is sent to the "emails" queue...
dispatch((new Job)->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 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.

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,
],
```

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- Redis: `predis/predis ~1.0`

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. You may generate a new queued job using the Artisan CLI:

```
php artisan make:job SendReminderEmail
```

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.

Class Structure

Job classes are very simple, normally containing only a `handle` method which is called when the job is processed by the queue. To get started, let's take a look at an example job class. 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\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param  Podcast  $podcast
     * @return void
     */
```

```
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  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 will be gracefully serialized and unserialized when the job is processing. If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance from the database. It's all totally transparent to your application and prevents issues that can arise from serializing full Eloquent model instances.

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

> {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.

Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` helper. The only argument you need to pass to the `dispatch` helper is an instance of the job:

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
```

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        dispatch(new ProcessPodcast($podcast));
    }
}
```

{tip} The `dispatch` helper provides the convenience of a short, globally available function, while also being extremely easy to test. Check out the Laravel testing documentation to learn more.

Delayed Dispatching

If you would like to delay the execution of a queued job, you may use the `delay` method on your job instance. The `delay` method is provided by the `Illuminate\Bus\Queueable` trait, which is included by default on all generated job classes. 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 Carbon\Carbon;
use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
```

```
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        $job = (new ProcessPodcast($podcast))
                    ->delay(Carbon::now()->addMinutes(10));

        dispatch($job);
    }
}
```

{note} The Amazon SQS queue service has a maximum delay time of 15 minutes.

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 on the job instance:

```
<?php

namespace App\Http\Controllers;

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...
```

```
        $job = (new ProcessPodcast($podcast))->onQueue('processing');

        dispatch($job);
    }
}
```

Dispatching To A Particular Connection

If you are working with multiple queue connections, you may specify which connection to push a job to. To specify the connection, use the `onConnection` method on the job instance:

```
<?php

namespace App\Http\Controllers;

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        $job = (new ProcessPodcast($podcast))->onConnection('sqs');

        dispatch($job);
    }
}
```

Of course, you may chain the `onConnection` and `onQueue` methods to specify the connection and the queue for a job:

```
$job = (new ProcessPodcast($podcast))
                ->onConnection('sqs')
                ->onQueue('processing');
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

One approach to specifying the maximum number of times a job may be attempted is via the `--tries` switch on the Artisan command line:

```
php artisan queue:work --tries=3
```

However, you may take a more granular approach by defining the maximum number of attempts on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the 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;
}
```

Timeout

Likewise, 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
```

However, 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;
}
```

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.

Running The Queue Worker

Laravel includes a queue worker that will 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.

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 only that queue:

```
php artisan queue:work redis --queue=emails
```

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should free 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.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` 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 pick up 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 "die" after they finish processing their current job so that no existing jobs are lost. Since the queue workers will die when the `queue:restart` command is executed, you should be running a process manager such as Supervisor to automatically restart the queue workers.

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 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. The `--timeout` option specifies how long the Laravel queue master process will wait before killing off a child queue worker that is processing a job. Sometimes a child queue process can become "frozen" for various reasons, such as an external HTTP call that is not responding. The `--timeout` option removes frozen processes that have exceeded that specified time limit:

```
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 given job is always killed before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

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 long 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
```

Supervisor Configuration

Installing Supervisor

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

```
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.

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 a `queue:work` process:

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

In this example, the `numprocs` directive will instruct Supervisor to run 8 `queue:work` processes and monitor all of them, automatically restarting them if they fail. Of course, you should change the `queue:work sqs` portion of the `command` directive to reflect your desired queue connection.

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 amount of attempts, it will be inserted into the `failed_jobs` database table. 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
```

Then, when running your queue worker, you should 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 be attempted indefinitely:

```
php artisan queue:work redis --tries=3
```

Cleaning Up After Failed Jobs

You may define a `failed` method directly on your job class, allowing you to perform job specific clean-up when a failure occurs. This is the perfect location to send an alert to your users or revert any actions performed by the job. The `Exception` that caused the job to fail will be passed to the `failed` method:

```
<?php

namespace App\Jobs;

use Exception;
use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param  Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
```

```
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }

    /**
     * The job failed to process.
     *
     * @param  Exception  $exception
     * @return void
     */
    public function failed(Exception $exception)
    {
        // Send user notification of failure, etc...
    }
}
```

Failed Job Events

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

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobFailed;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
```

```
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }

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

Retrying Failed Jobs

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

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, and failure time. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of `5`, issue the following command:

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

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
```

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

```
php artisan queue:flush
```

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 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
{
    /**
     * 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()
        });
    }

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

```
        //
    }
}
```

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:

```
Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

Task Scheduling

Introduction

In the past, you may have generated a Cron 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 add additional Cron entries.

Laravel's command scheduler allows you to fluently and expressively define your command schedule within Laravel 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.

Starting The Scheduler

When using the scheduler, you only need to add the following Cron entry to your server. 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:

```
* * * * * php /path-to-your-project/artisan schedule:run >> /dev/null 2>&1
```

This Cron will call the Laravel command scheduler every minute. When the `schedule:run` command is executed, Laravel will evaluate your scheduled tasks and runs the tasks that are due.

Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of the `App\Console\Kernel` class. To get started, let's look at an example of scheduling a task. 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 DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        \App\Console\Commands\Inspire::class,
    ];

    /**
     * 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 `Closure` calls, you may also schedule Artisan commands and operating system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class:

```
$schedule->command('emails:send --force')->daily();

$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

The `exec` command may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

Of course, there are a variety of schedules you may assign to your task:

Method	Description
`->cron('* * * * * *');`	Run the task on a custom Cron schedule
`->everyMinute();`	Run the task every minute
`->everyFiveMinutes();`	Run the task every five minutes
`->everyTenMinutes();`	Run the task every ten minutes
`->everyThirtyMinutes();`	Run the task every thirty minutes
`->hourly();`	Run the task every hour
`->hourlyAt(17);`	Run the task every hour at 17 mins past the hour
`->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 week
`->monthly();`	Run the task every month
`->monthlyOn(4, '15:00');`	Run the task every month on the 4th at 15:00
`->quarterly();`	Run the task every quarter
`->yearly();`	Run the task every year
`->timezone('America/New_York');`	Set the timezone

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, to 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');
```

Below is a list of the additional schedule constraints:

Method	Description
`->weekdays();`	Limit the task to weekdays
`->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
`->between($start, $end);`	Limit the task to run between start and end times
`->when(Closure);`	Limit the task based on a truth test

Between Time Constraints

The `between` method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('reminders: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('reminders: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`.

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.

Maintenance Mode

Laravel's scheduled tasks will not run when Laravel 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 use the `evenInMaintenanceMode` method:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

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 e-mail the output to an e-mail address of your choice. Before e-mailing the output of a task, you should configure Laravel's e-mail services:

```
$schedule->command('foo')
         ->daily()
         ->sendOutputTo($filePath)
         ->emailOutputTo('foo@example.com');
```

> {note} The `emailOutputTo`, `sendOutputTo` and `appendOutputTo` methods are exclusive to the `command` method and are not supported for `call`.

Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is complete:

```
$schedule->command('emails:send')
         ->daily()
         ->before(function () {
             // Task is about to start...
         })
         ->after(function () {
             // Task is complete...
         });
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is complete. This method is useful for notifying an external service, such as Laravel Envoyer, that your scheduled task is commencing or has finished execution:

```
$schedule->command('emails:send')
         ->daily()
```

```
        ->pingBefore($url)
        ->thenPing($url);
```

Using either the `pingBefore($url)` or `thenPing($url)` feature requires the Guzzle HTTP library. You can add Guzzle to your project using the Composer package manager:

```
composer require guzzlehttp/guzzle
```

Database: Getting Started

Introduction

Laravel makes interacting with databases extremely simple across a variety of database backends using either raw SQL, the fluent query builder, and the Eloquent ORM. Currently, Laravel supports four databases:

- MySQL
- Postgres
- SQLite
- SQL Server

Configuration

The database configuration for your application is located at `config/database.php`. In this file you may define all of your database connections, as well as specify which connection should be used by default. Examples for most of the supported database systems are provided in this file.

By default, Laravel's sample environment configuration is ready to use with Laravel Homestead, which is a convenient virtual machine for doing Laravel development on your local machine. Of course, you are free to modify this configuration as needed for your local database.

SQLite Configuration

After creating a new SQLite database using a command such as `touch database/database.sqlite`, you can easily configure your environment variables to point to this newly created database by using the database's absolute path:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

SQL Server Configuration

Laravel supports SQL Server out of the box; however, you will need to add the connection configuration for the database to your `config/database.php` configuration file:

```
'sqlsrv' => [
    'driver' => 'sqlsrv',
    'host' => env('DB_HOST', 'localhost'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'charset' => 'utf8',
    'prefix' => '',
],
```

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',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
```

```
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix'    => '',
],
```

Note that two keys have been added to the configuration array: `read` and `write`. Both of these 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` array.

You only need to place items in the `read` and `write` arrays if you wish to override the values from the main array. So, in this case, `192.168.1.1` will be used as the host for the "read" connection, while `192.168.1.2` 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.

Using Multiple Database Connections

When using multiple connections, you may access each connection via the `connection` method on the `DB` facade. The `name` passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file:

```
$users = DB::connection('foo')->select(...);
```

You may also access the raw, underlying PDO instance using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Running Raw 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 query, you may use the `select` method on the `DB` facade:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;
```

```
class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return 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 raw 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, allowing you to access the values of the results:

```
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 takes the raw SQL query as its first argument and bindings as its second argument:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

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 will be returned:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

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:

```
$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');
```

Listening For Query Events

If you would like to receive each SQL query executed by your application, you may use the `listen` method. This method is useful for logging queries or debugging. You may register your query listener in a service provider:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql
            // $query->bindings
            // $query->time
        });
    }

    /**
```

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

Database Transactions

You may use the `transaction` method on 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:

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

Handling Deadlocks

The `transaction` method accepts an optional second argument which defines the number of times a transaction should be reattempted when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
}, 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 on the `DB` facade:

```
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} Using the `DB` facade's transaction methods also controls transactions for the query builder and Eloquent ORM.

Database: Query Builder

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 on all 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 strings being passed as bindings.

Retrieving Results

Retrieving All Rows From A Table

You may use the `table` method on 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 get the results using the `get` method:

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return 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 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:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from the database table, you may use the `first` method. This method will return a single `StdClass` object:

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

If you don't even 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');
```

Retrieving A List Of Column Values

If you would like to retrieve a Collection containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve a Collection of role titles:

```
$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may also specify a custom key column for the returned Collection:

```
$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

Chunking Results

If you need to work with thousands of database records, consider using the `chunk` method. This method retrieves a small chunk of the results at a time and feeds each chunk into a `Closure` for processing. This method is very useful for writing Artisan commands that process thousands of records. For example, let's work with the entire `users` table in chunks of 100 records at a time:

```
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;
});
```

Aggregates

The query builder also provides a variety of aggregate methods such as `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Selects

Specifying A Select Clause

Of course, 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:

```
$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 use a raw expression in a query. These expressions will be injected into the query as strings, so be careful not to create any SQL injection points! To create a raw expression, you may use the `DB::raw` method:

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

Joins

Inner Join Clause

The query builder may also be used to write join statements. 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. Of course, as you can see, you can join to multiple tables in a single query:

```
$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 Clause

If you would like to perform a "left join" instead of an "inner join", use the `leftJoin` method. The `leftJoin` method has the same signature as the `join` method:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Cross Join Clause

To perform a "cross join" use the `crossJoin` method with the name of the table you wish to cross join to. Cross joins generate a cartesian product between the first table and the joined table:

```
$users = DB::table('sizes')
            ->crossJoin('colours')
            ->get();
```

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a `Closure` as the second argument into the `join` method. The `Closure` will receive a `JoinClause` object 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" style clause on your joins, you may use the `where` and `orWhere` methods on a join. 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();
```

Unions

The query builder also provides a quick way to "union" two queries together. For example, you may create an initial query and use the `union` method to union it with a second query:

```
$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

> {tip} The `unionAll` method is also available and has the same method signature as `union`.

Where Clauses

Simple Where Clauses

You may use the `where` method on a query builder instance to add `where` clauses to the query. The most basic call to `where` 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. Finally, the third argument is the value to evaluate against the column.

For example, here is a query that verifies the value of the "votes" column is equal to 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

For convenience, if you simply want to verify that a column is equal to a given value, you may pass the value directly as the second argument to the `where` method:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Of course, you may use a variety of other operators when writing a `where` clause:

```
$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:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

Or Statements

You may chain where constraints together as well as add `or` clauses to the query. The `orWhere` method accepts the same arguments as the `where` method:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Additional Where Clauses

whereBetween

The `whereBetween` method verifies that a column's value is between two values:

```
$users = DB::table('users')
                    ->whereBetween('votes', [1, 100])->get();
```

whereNotBetween

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

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();
```

whereNull / whereNotNull

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

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 of a year:

```
$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 a 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();
```

whereColumn

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 method:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

The `whereColumn` method can also be passed an array of multiple conditions. These conditions will be joined using the `and` operator:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at']
                ])->get();
```

Parameter Grouping

Sometimes you may need to create more advanced where clauses such as "where exists" clauses or nested parameter groupings. The Laravel query builder can handle these as well. To get started, let's look at an example of grouping constraints within parenthesis:

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function ($query) {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

As you can see, passing a `Closure` into the `orWhere` 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' or (votes > 100 and title <> 'Admin')
```

Where Exists Clauses

The `whereExists` method allows you to write `where exists` SQL clauses. The `whereExists` method accepts a `Closure` argument, which will receive a query builder instance allowing you to define the query that should be placed inside of the "exists" clause:

```
DB::table('users')
            ->whereExists(function ($query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('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
)
```

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 and Postgres. To query a JSON column, use the `->` operator:

```
$users = DB::table('users')
                ->where('options->language', 'en')
                ->get();

$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

Ordering, Grouping, Limit, & Offset

orderBy

The `orderBy` method allows you to sort the result of the query by a given column. The first argument to the `orderBy` method should be the column you wish to sort by, while the second argument controls the direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

latest / oldest

The `latest` and `oldest` methods allow you to easily order results by date. By default, result will be ordered by the `created_at` column. Or, you may pass the column name that you wish to sort by:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

inRandomOrder

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();
```

groupBy / having / havingRaw

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();
```

The `havingRaw` method may be used to set a raw string as the value of the `having` clause. For example, we can find all of the departments with sales greater than $2,500:

```
$users = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > 2500')
                ->get();
```

skip / take

To limit the number of results returned from the query, or to skip a given number of results in the query, you may use the `skip` and `take` methods:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternatively, you may use the `limit` and `offset` methods:

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

Conditional Clauses

Sometimes you may want clauses to apply to a query only when something else is true. For instance you may only want to apply a `where` statement if a given input value is present on the incoming request. You may accomplish this using the `when` method:

```
$role = $request->input('role');

$users = DB::table('users')
                ->when($role, function ($query) use ($role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

The `when` method only executes the given Closure when the first parameter is `true`. If the first parameter is `false`, the Closure will not be executed.

You may pass another Closure as the third parameter to the `when` method. This Closure will execute if the first parameter evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default sorting of a query:

```
$sortBy = null;

$users = DB::table('users')
                ->when($sortBy, function ($query) use ($sortBy) {
                    return $query->orderBy($sortBy);
                }, function ($query) {
                    return $query->orderBy('name');
                })
                ->get();
```

Inserts

The query builder also provides an `insert` method for inserting records into the database table. The `insert` method accepts an array of column names and values:

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

You may even insert several records into the table with a single call to `insert` by passing an array of arrays. Each array represents a row to be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

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 sequence name as the second parameter to the `insertGetId` method.

Updates

Of course, 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 containing the columns to be updated. You may constrain the `update` query using `where` clauses:

```
DB::table('users')
            ->where('id', 1)
            ->update(['votes' => 1]);
```

Updating JSON Columns

When updating a JSON column, you should use `->` syntax to access the appropriate key in the JSON object. This operation is only supported on databases that support JSON columns:

```
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. This is simply a shortcut, providing a more expressive and terse interface compared to manually writing the `update` statement.

Both of these methods accept at least one argument: the column to modify. A second argument may optionally be passed to control 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']);
```

Deletes

The query builder may also be used to delete records from the table via the `delete` method. 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 the entire table, which will remove all rows and reset the auto-incrementing ID to zero, you may use the `truncate` method:

```
DB::table('users')->truncate();
```

Pessimistic Locking

The query builder also includes a few functions to help you do "pessimistic locking" on your `select` statements. To run the statement with a "shared lock", you may use the `sharedLock` method on a query. A shared lock prevents the selected rows from being modified until your transaction commits:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the rows from being modified or from being selected with another shared lock:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

Database: Pagination

Introduction

In other frameworks, pagination can be very painful. Laravel's paginator is integrated with the query builder and Eloquent ORM and provides convenient, easy-to-use pagination of database results out of the box. The HTML generated by the paginator is compatible with the Bootstrap CSS framework.

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 proper 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. Of course, 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 Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show all of the users for the application.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);

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

{note} Currently, pagination operations that use a `groupBy` statement cannot be executed efficiently by Laravel. If you need to use a `groupBy` with a paginated result set, it is recommended that you query the database and create a paginator manually.

"Simple Pagination"

If you only need to display simple "Next" and "Previous" links in your pagination view, you may use the `simplePaginate` method to perform a more efficient query. This is very useful for large datasets when you do not need to display a link for each page number when rendering your view:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries. In this example, we will paginate the `User` model with `15` items per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
$users = App\User::paginate(15);
```

Of course, you may call `paginate` 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. 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 does require 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 and Eloquent, 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.

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 Bootstrap CSS framework.

Customizing The Paginator URI

The `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/custom/url?page=N`, you should pass `custom/url` to the `withPath` method:

```
Route::get('users', function () {
    $users = App\User::paginate(15);

    $users->withPath('custom/url');

    //
});
```

Appending To Pagination Links

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`:

```
{{ $users->appends(['sort' => 'votes'])->links() }}
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the `fragment` method. For example, to append `#foo` to the end of each pagination link, make the following call to the `fragment` method:

```
{{ $users->fragment('foo')->links() }}
```

Converting Results To JSON

The Laravel paginator result 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 simply returning it from a route or controller action:

```
Route::get('users', function () {
    return App\User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The actual result objects will be 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,
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "from": 1,
   "to": 15,
   "data":[
        {
            // Result Object
        },
        {
            // Result Object
        }
   ]
}
```

Customizing The Pagination View

By default, the views rendered to display the pagination links are compatible with the Bootstrap CSS framework. However, if you are not using Bootstrap, you are free to define your own views to render these links. When calling the `links` method on a paginator instance, pass the view name as the first argument to the method:

```
{{ $paginator->links('view.name') }}

// Passing 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 the `resources/views/vendor/pagination` directory. The `default.blade.php` file within this directory corresponds to the default pagination view. Simply edit this file to modify the pagination HTML.

Paginator Instance Methods

Each paginator instance provides additional pagination information via the following methods:

- `$results->count()`
- `$results->currentPage()`
- `$results->firstItem()`
- `$results->hasMorePages()`
- `$results->lastItem()`
- `$results->lastPage() (Not available when using simplePaginate)`
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total() (Not available when using simplePaginate)`
- `$results->url($page)`

Database: Migrations

Introduction

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema. If you have ever had to tell a teammate to manually add a column to their local database schema, 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.

Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Laravel to determine the order of the migrations.

The `--table` and `--create` options may also be used to indicate the name of the table and whether the migration will be creating a new table. These options simply pre-fill the generated migration stub file with the specified table:

```
php artisan make:migration create_users_table --create=users

php artisan make:migration add_votes_to_users_table --table=users
```

If you would like to specify a custom output path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The given path should be relative to your application's base path.

Migration 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 simply 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, this migration example creates a `flights` table:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
```

```
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('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
```

> {note} If you are using the Homestead virtual machine, you should run this command from within your virtual machine.

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 rollback the latest migration operation, you may use the `rollback` command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may rollback a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will rollback 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
```

Rollback & Migrate In 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 rollback & re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will rollback & re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

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:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Of course, 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 easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
    //
}
```

Connection & Storage Engine

If you want to perform a schema operation on a database connection that is not your default connection, use the `connection` method:

```
Schema::connection('foo')->create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

You may use the `engine` property on the schema builder to define the table's storage engine:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    $table->increments('id');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
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 a `Blueprint` instance you may use to add columns to the table:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email');
});
```

Available Column Types

Of course, the schema builder contains a variety of column types that you may specify when building your tables:

Command	Description
`$table->bigIncrements('id');`	Incrementing ID (primary key) using a "UNSIGNED BIG INTEGER" equivalent.
`$table->bigInteger('votes');`	BIGINT equivalent for the database.
`$table->binary('data');`	BLOB equivalent for the database.
`$table->boolean('confirmed');`	BOOLEAN equivalent for the database.
`$table->char('name', 4);`	CHAR equivalent with a length.
`$table->date('created_at');`	DATE equivalent for the database.
`$table->dateTime('created_at');`	DATETIME equivalent for the database.

Command	Description
`$table->dateTimeTz('created_at');`	DATETIME (with timezone) equivalent for the database.
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale.
`$table->double('column', 15, 8);`	DOUBLE equivalent with precision, 15 digits in total and 8 after the decimal point.
`$table->enum('choices', ['foo', 'bar']);`	ENUM equivalent for the database.
`$table->float('amount', 8, 2);`	FLOAT equivalent for the database, 8 digits in total and 2 after the decimal point.
`$table->increments('id');`	Incrementing ID (primary key) using a "UNSIGNED INTEGER" equivalent.
`$table->integer('votes');`	INTEGER equivalent for the database.
`$table->ipAddress('visitor');`	IP address equivalent for the database.
`$table->json('options');`	JSON equivalent for the database.

Command	Description
`$table->jsonb('options');`	JSONB equivalent for the database.
`$table->longText('description');`	LONGTEXT equivalent for the database.
`$table->macAddress('device');`	MAC address equivalent for the database.
`$table->mediumIncrements('id');`	Incrementing ID (primary key) using a "UNSIGNED MEDIUM INTEGER" equivalent.
`$table->mediumInteger('numbers');`	MEDIUMINT equivalent for the database.
`$table->mediumText('description');`	MEDIUMTEXT equivalent for the database.
`$table->morphs('taggable');`	Adds unsigned INTEGER `taggable_id` and STRING `taggable_type`.
`$table->nullableMorphs('taggable');`	Nullable versions of the `morphs()` columns.
`$table->nullableTimestamps();`	Nullable versions of the `timestamps()` columns.
`$table->rememberToken();`	Adds `remember_token` as VARCHAR(100) NULL.

Command	Description
`$table->smallIncrements('id');`	Incrementing ID (primary key) using a "UNSIGNED SMALL INTEGER" equivalent.
`$table->smallInteger('votes');`	SMALLINT equivalent for the database.
`$table->softDeletes();`	Adds nullable `deleted_at` column for soft deletes.
`$table->string('email');`	VARCHAR equivalent column.
`$table->string('name', 100);`	VARCHAR equivalent with a length.
`$table->text('description');`	TEXT equivalent for the database.
`$table->time('sunrise');`	TIME equivalent for the database.
`$table->timeTz('sunrise');`	TIME (with timezone) equivalent for the database.
`$table->tinyInteger('numbers');`	TINYINT equivalent for the database.
`$table->timestamp('added_on');`	TIMESTAMP equivalent for the database.
`$table->timestampTz('added_on');`	TIMESTAMP (with timezone) equivalent for the database.

Command	Description
`$table->timestamps();`	Adds nullable `created_at` and `updated_at` columns.
`$table->timestampsTz();`	Adds nullable `created_at` and `updated_at` (with timezone) columns.
`$table->unsignedBigInteger('votes');`	Unsigned BIGINT equivalent for the database.
`$table->unsignedInteger('votes');`	Unsigned INT equivalent for the database.
`$table->unsignedMediumInteger('votes');`	Unsigned MEDIUMINT equivalent for the database.
`$table->unsignedSmallInteger('votes');`	Unsigned SMALLINT equivalent for the database.
`$table->unsignedTinyInteger('votes');`	Unsigned TINYINT equivalent for the database.
`$table->uuid('id');`	UUID equivalent for the database.

Column Modifiers

In addition to the column types listed above, there are several column "modifiers" you may use while adding a column to a database table. For example, to make the column "nullable", you may use the `nullable` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
```

```
});
```

Below is a list of all the available column modifiers. This list does not include the index modifiers:

Modifier	Description
`->after('column')`	Place the column "after" another column (MySQL Only)
`->comment('my comment')`	Add a comment to a column
`->default($value)`	Specify a "default" value for the column
`->first()`	Place the column "first" in the table (MySQL Only)
`->nullable()`	Allow NULL values to be inserted into the column
`->storedAs($expression)`	Create a stored generated column (MySQL Only)
`->unsigned()`	Set `integer` columns to `UNSIGNED`
`->virtualAs($expression)`	Create a virtual generated column (MySQL Only)

Modifying Columns

Prerequisites

Before modifying a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file. The Doctrine DBAL library is used to determine the current state of the column and create the SQL queries needed to make the specified adjustments to the column:

```
composer require doctrine/dbal
```

Updating Column Attributes

The `change` method allows you to modify some existing column types to a new type or modify the column's attributes. 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:

```
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 not be "changed": char, double, enum, mediumInteger, timestamp, tinyInteger, ipAddress, json, jsonb, macAddress, mediumIncrements, morphs, nullableMorphs,

nullableTimestamps, softDeletes, timeTz, timestampTz, timestamps, timestampsTz, unsignedMediumInteger, unsignedTinyInteger, uuid.

Renaming Columns

To rename a column, you may use the `renameColumn` method on the Schema builder. Before renaming a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

{note} Renaming any column in a table that also has a column of type `enum` is not currently supported.

Dropping Columns

To drop a column, use the `dropColumn` method on the Schema builder. Before dropping columns from a SQLite database, you will need to add the `doctrine/dbal` dependency to your `composer.json` file and run the `composer update` command in your terminal to install the library:

```
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.

Indexes

Creating Indexes

The schema builder supports several types of indexes. First, let's look at an example that specifies a column's values should be unique. To create the index, we can simply chain the `unique` method onto the column definition:

```
$table->string('email')->unique();
```

Alternatively, you may create the index after defining the column. For example:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound index:

```
$table->index(['account_id', 'created_at']);
```

Laravel will automatically generate a reasonable index name, but you may pass a second argument to the method to specify the name yourself:

```
$table->index('email', 'my_index_name');
```

Available Index Types

Command	Description
`$table->primary('id');`	Add a primary key.
`$table->primary(['first', 'last']);`	Add composite keys.
`$table->unique('email');`	Add a unique index.
`$table->unique('state', 'my_index_name');`	Add a custom index name.
`$table->unique(['first', 'last']);`	Add a composite unique index.
`$table->index('state');`	Add a basic index.

Index Lengths & MySQL / MariaDB

Laravel uses the `utf8mb4` character set by default, which includes support for storing "emojis" in the database. 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 this by calling the `Schema::defaultStringLength` method within your `AppServiceProvider`:

```
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.

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns a reasonable name to the indexes. Simply concatenate 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.

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 key 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:

```
Schema::table('posts', function (Blueprint $table) {
    $table->integer('user_id')->unsigned();

    $table->foreign('user_id')->references('id')->on('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreign('user_id')
```

```
->references('id')->on('users')
->onDelete('cascade');
```

To drop a foreign key, you may use the `dropForeign` method. Foreign key constraints use the same naming convention as indexes. So, we will concatenate the table name and the columns in the constraint then suffix the name with "_foreign":

```
$table->dropForeign('posts_user_id_foreign');
```

Or, you may pass an array value which will automatically use the conventional constraint name when dropping:

```
$table->dropForeign(['user_id']);
```

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

Database: Seeding

- Introduction
- Writing Seeders
  - Using Model Factories
  - Calling Additional Seeders
- Running Seeders

Introduction

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in the `database/seeds` directory. Seed classes may have any name you wish, but probably should follow some sensible convention, such as `UsersTableSeeder`, etc. 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.

Writing Seeders

To generate a seeder, execute the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeds` directory:

```
php artisan make:seeder UsersTableSeeder
```

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

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

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. Once you have defined your factories, you may use the `factory` helper function to insert records into your database.

For example, let's create 50 users and attach a relationship to each user:

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
```

```
    factory(App\User::class, 50)->create()->each(function ($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}
```

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 overwhelmingly large. Simply pass the name of the seeder class you wish to run:

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
}
```

Running Seeders

Once you have written your seeder classes, you may use the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `DatabaseSeeder` class, which may be used to call 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=UsersTableSeeder
```

You may also seed your database using the `migrate:refresh` command, which will also rollback and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:refresh --seed
```

Redis

- Introduction

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, you will need to install the `predis/predis` package via Composer:

```
composer require predis/predis
```

Alternatively, you may install the PhpRedis PHP extension via PECL. The extension is more complex to install but may yield better performance for applications that make heavy use of Redis.

Configuration

The Redis configuration for your application is located in 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' => 'predis',

    'default' => [
        'host' => env('REDIS_HOST', 'localhost'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => 0,
    ],

],
```

The default server configuration should suffice for development. However, you are free to modify this array based on your environment. Each Redis server defined in your configuration file is required to have a name, host, and port.

Configuring Clusters

If your application is utilizing a cluster of Redis servers, you should define these clusters within a `clusters` key of your Redis configuration:

```
'redis' => [

    'client' => 'predis',

    '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, note that client-side sharding does not handle failover; therefore, is primarily suited for cached data that is available from another primary data store. If you would like to use native Redis clustering, you should specify this in the `options` key of your Redis configuration:

```
'redis' => [

    'client' => 'predis',

    'options' => [
        'cluster' => 'redis',
    ],

    'clusters' => [
        // ...
    ],

],
```

Predis

In addition to the default `host`, `port`, `database`, and `password` server configuration options, Predis supports additional connection parameters that may

be defined for each of your Redis servers. To utilize these additional configuration options, simply add them 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_write_timeout' => 60,
],
```

PhpRedis

> {note} If you have the PhpRedis PHP extension installed via PECL, you will need to rename the `Redis` alias in your `config/app.php` configuration file.

To utilize the PhpRedis extension, you should change the `client` option of your Redis configuration to `phpredis`. This option is found in your `config/database.php` configuration file:

```
'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: `persistent`, `prefix`, `read_timeout` and `timeout`. 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,
],
```

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

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

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

Of course, as mentioned above, you may call any of the Redis commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server, so simply pass the arguments the Redis command expects:

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may also pass commands to the server using the `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

You may get a Redis instance by calling the `Redis::connection` method:

```
$redis = Redis::connection();
```

This will give you an instance of the default Redis server. You may also pass the connection or cluster name to the `connection` method to get a specific server or

cluster as defined in your Redis configuration:

```
$redis = Redis::connection('my-connection');
```

Pipelining Commands

Pipelining should be used when you need to send many commands to the server in one operation. 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 executed within a single operation:

```
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:

```
Route::get('publish', function () {
    // Route logic...

    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

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 callback `Closure`:

```
Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {
    echo $message;
});
```

Eloquent: Getting Started

- Introduction
- Defining Models

Introduction

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.

Before getting started, be sure to configure a database connection in `config/database.php`. For more information on configuring your database, check out the documentation.

Defining Models

To get started, let's create an Eloquent model. Models typically live in the `app` directory, but you are free to place them anywhere that can be autoloaded according to your `composer.json` file. All Eloquent models extend `Illuminate\Database\Eloquent\Model` class.

The easiest way to create a model instance is using the `make:model` Artisan command:

```
php artisan make:model User
```

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 User --migration

php artisan make:model User -m
```

Eloquent Model Conventions

Now, let's look at an example `Flight` model, which we will use to retrieve and store information from our `flights` database table:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

Note that we did not tell Eloquent which table to use for 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. You may specify a custom table by defining a `table` property on your model:

```
<?php

namespace App;

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 table has a primary key column named `id`. You may define a `$primaryKey` property to override this convention.

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that by default the primary key will be cast to an `int` automatically. If you wish to use a non-incrementing or a non-numeric primary key you must set the public `$incrementing` property on your model to `false`.

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to `false`:

```
<?php

namespace App;

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 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;

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 set the `CREATED_AT` and `UPDATED_AT` constants in your model:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
}
```

Database Connection

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the `$connection` property:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```
<?php

use App\Flight;
```

```
$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

Adding Additional Constraints

The Eloquent `all` method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the `get` method to retrieve the results:

```
$flights = App\Flight::where('active', 1)
               ->orderBy('name', 'desc')
               ->take(10)
               ->get();
```

> {tip} Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

Collections

For Eloquent methods like `all` and `get` which retrieve multiple results, an instance of `Illuminate\Database\Eloquent\Collection` will be returned. The `Collection` class provides a variety of helpful methods for working with your Eloquent results:

```
$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

Of course, you may also simply loop over the collection like an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

If you need to process thousands of Eloquent records, use the `chunk` command. The `chunk` method will retrieve a "chunk" of Eloquent models, feeding them to a given `Closure` for processing. Using the `chunk` method will conserve memory when working with large result sets:

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called 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.

Using Cursors

The `cursor` method allows you to iterate through your database records using a cursor, which will only execute a single query. When processing large amounts of data, the `cursor` method may be used to greatly reduce your memory usage:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {
    //
}
```

Retrieving Single Models / Aggregates

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using `find` and `first`. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

You may also call the `find` method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

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, a `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$model = App\Flight::findOrFail(1);
```

```
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a 404 HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return 404 responses when using these methods:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

Retrieving Aggregates

You may also use the count, sum, max, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Inserts

To create a new record in the database, simply create a new model instance, set attributes on the model, then call the save method:

```
<?php

namespace App\Http\Controllers;

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

class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...
```

```
        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

In this example, we simply assign the `name` parameter from the incoming HTTP request to the `name` attribute of the `App\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The `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.

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, set any attributes you wish to update, and then call the `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Mass Updates

Updates can also be performed against any number of 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:

```
App\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 `saved` 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.

Mass Assignment

You may also use the `create` method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a `fillable` or `guarded` attribute on the model, as all Eloquent models protect against mass-assignment by default.

A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed into 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;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database. The `create` method returns the saved model instance:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```
$flight->fill(['name' => 'Flight 22']);
```

Guarding Attributes

While `$fillable` serves as a "white list" of attributes that should be mass assignable, you may also choose to use `$guarded`. The `$guarded` property

should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, `$guarded` functions like a "black list". Of course, you should use either `$fillable` or `$guarded` - not both. In the example below, all attributes **except for `price`** will be mass assignable:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

If you would like to make all attributes mass assignable, you may define the `$guarded` property as an empty array:

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Other Creation Methods

`firstOrCreate`/ `firstOrNew`

There are two other methods you may use to create models by mass assigning attributes: `firstOrCreate` and `firstOrNew`. 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 given attributes.

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 call `save` manually to persist it:

```
// Retrieve flight by name, or create it if it doesn't exist...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// Retrieve flight by name, or create it with the name and delayed attributes...
$flight = App\Flight::firstOrCreate(
    ['name' => 'Flight 10'], ['delayed' => 1]
);

// Retrieve by name, or instantiate...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// Retrieve by name, or instantiate with the name and delayed attributes...
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'], ['delayed' => 1]
);
```

`updateOrCreate`

You may also come across situations where you want to update an existing model or create a new model if none exists. Laravel provides an `updateOrCreate` method to do this in one step. Like the `firstOrCreate` method, `updateOrCreate` persists the model, so there's no need to call `save()`:

```
// If there's a flight from Oakland to San Diego, set the price to $99.
// If no matching model exists, create one.
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99]
);
```

Deleting Models

To delete a model, call the `delete` method on a model instance:

```
$flight = App\Flight::find(1);

$flight->delete();
```

Deleting An Existing Model By 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 retrieving it. To do so, call the `destroy` method:

```
App\Flight::destroy(1);

App\Flight::destroy([1, 2, 3]);

App\Flight::destroy(1, 2, 3);
```

Deleting Models By Query

Of course, you may also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

> {note} When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be fired 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 and inserted into the database. If a model has a non-null `deleted_at` value, the model has been soft deleted. To enable soft deletes for a model, use the `Illuminate\Database\Eloquent\SoftDeletes` trait on the model and add the `deleted_at` column to your `$dates` property:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;

    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
```

```
}
```

Of course, you should add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. And, 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, use the `trashed` method:

```
if ($flight->trashed()) {
    //
}
```

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 appear in a result set using the `withTrashed` method on the query:

```
$flights = App\Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

The `withTrashed` method may also be used on a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = App\Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model into an active state, use the `restore` method on a model instance:

```
$flight->restore();
```

You may also use the `restore` method in a query to quickly restore multiple models. Again, like other "mass" operations, this will not fire any model events for the models that are restored:

```
App\Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

Like the `withTrashed` method, the `restore` method may also be used on relationships:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. To permanently remove a soft deleted model from the database, use the `forceDelete` method:

```
// Force deleting a single model instance...
$flight->forceDelete();

// Force deleting all related models...
$flight->history()->forceDelete();
```

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 pull "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. Define a class that implements the `Illuminate\Database\Eloquent\Scope` interface. This interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints to the query as needed:

```
<?php

namespace App\Scopes;
```

```
use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope 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('age', '>', 200);
    }
}
```

{tip} There is not a predefined folder for scopes in a default Laravel application, so feel free to make your own `Scopes` folder within your Laravel application's `app` directory.

Applying Global Scopes

To assign a global scope to a model, you should override a given model's `boot` method and use the `addGlobalScope` method:

```
<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();
```

```
        static::addGlobalScope(new AgeScope);
    }
}
```

After adding the scope, a query to `User::all()` will produce the following SQL:

```
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:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function (Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. The method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AgeScope::class)->get();
```

If you would like to remove several or even all of the 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 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, simply prefix an Eloquent model method with `scope`.

Scopes should always return a query builder instance:

```
<?php

namespace App;

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 do not need to include the `scope` prefix when calling the method. You can even chain calls to various scopes, for example:

```
$users = App\User::popular()->active()->orderBy('created_at')->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. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App;

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);
    }
}
```

Now, you may pass the parameters when calling the scope:

```
$users = App\User::ofType('admin')->get();
```

Events

Eloquent models fire several events, allowing you to hook into the following points in a model's lifecycle: `creating`, `created`, `updating`, `updated`, `saving`,

`saved`, `deleting`, `deleted`, `restoring`, `restored`. Events allow you to easily execute code each time a specific model class is saved or updated in the database.

Whenever a new model is saved for the first time, the `creating` and `created` events will fire. If a model already existed in the database and the `save` method is called, the `updating` / `updated` events will fire. However, in both cases, the `saving` / `saved` events will fire.

To get started, define an `$events` property on your Eloquent model that maps various points of the Eloquent model's lifecycle to your own event classes:

```
<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $events = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

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. Observers classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the model as their only argument. Laravel does not include a default directory for observers, so you may create any directory you like to house your observer classes:

```
<?php

namespace App\Observers;
```

```
use App\User;

class UserObserver
{
    /**
     * Listen to the User created event.
     *
     * @param  User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Listen to the User deleting event.
     *
     * @param  User  $user
     * @return void
     */
    public function deleting(User $user)
    {
        //
    }
}
```

To register an observer, use the `observe` method on the model you wish to observe. You may register observers in the `boot` method of one of your service providers. In this example, we'll register the observer in the `AppServiceProvider`:

```
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
```

```
    public function boot()
    {
        User::observe(UserObserver::class);
    }

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

Eloquent: Relationships

- Introduction
- Defining Relationships
  - One To One
  - One To Many
  - One To Many (Inverse)
  - Many To Many
  - Has Many Through
  - Polymorphic Relations
  - Many To Many Polymorphic Relations
- Querying Relations
  - Relationship Methods Vs. Dynamic Properties
  - Querying Relationship Existence
  - Querying Relationship Absence
  - Counting Related Models
- 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 several different types of relationships:

- One To One
- One To Many
- Many To Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful query builders, defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional 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.

One To One

A one-to-one relationship is a very basic relation. For example, a `User` model might be associated with one `Phone`. To define this relationship, we place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
```

```
    {
        return $this->hasOne('App\Phone');
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model. 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 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('App\Phone', 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the `id` (or the custom `$primaryKey`) 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 value other than `id`, you may pass a third argument to the `hasOne` method specifying your custom key:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User`. Now, 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;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
```

```
}
```

In the example above, Eloquent will try to match the `user_id` from the `Phone` model to an `id` on the `User` model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. 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('App\User', 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

One To Many

A "one-to-many" relationship is used to define relationships where a single model owns any amount of other 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 placing a function on your Eloquent model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
```

```
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column on the `Comment` model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with `_id`. So, for this example, Eloquent will assume the foreign key on the `Comment` model is `post_id`.

Once the relationship has been defined, we can access the collection of comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic properties", we can access relationship methods as if they were defined as properties on the model:

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Of course, since all relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the `comments` method and continuing to chain conditions onto the query:

```
$comments = App\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('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

One To Many (Inverse)

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 function on the child model which calls the `belongsTo` method:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
```

```
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Once the relationship has been defined, we can retrieve the `Post` model for a `Comment` by accessing the `post` "dynamic property":

```
$comment = App\Comment::find(1);

echo $comment->post->title;
```

In the example above, Eloquent will try to match the `post_id` from the `Comment` model to an `id` on the `Post` model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. However, if the foreign key on the `Comment` model is not `post_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to 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('App\Post', 'foreign_key', 'other_key');
}
```

Many To Many

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of such a relationship is a user with many roles,

where the roles are also shared by other users. For example, many users may have the role of "Admin". 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 the `user_id` and `role_id` columns.

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. For example, let's define the `roles` method on our `User` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic property:

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```

Of course, like all other relationship types, you may call the `roles` method to continue chaining query constraints onto the relationship:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

As mentioned previously, to determine the table name of the relationship's joining 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('App\Role', 'role_user');
```

In addition to customizing the name of the joining 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('App\Role', 'role_user', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the inverse of a many-to-many relationship, you simply place another call to `belongsToMany` on your related model. To continue our user roles example, let's define the `users` method on the `Role` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` counterpart, with the exception of simply referencing the `App\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` object has many `Role` objects that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```
$user = App\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, and may be used like any other Eloquent model.

By default, only the model keys will be present on the `pivot` object. If your pivot table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

If you want your pivot table to have automatically maintained `created_at` and `updated_at` timestamps, use the `withTimestamps` method on the relationship definition:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Filtering Relationships Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` using the `wherePivot` and `wherePivotIn` methods when defining the relationship:

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your relationship, you may call the `using` method when defining the relationship. All custom models used to represent intermediate tables of relationships must extend the `Illuminate\Database\Eloquent\Relations\Pivot` class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\UserRole');
    }
}
```

Has Many Through

The "has-many-through" relationship provides a convenient shortcut for accessing distant relations via an intermediate relation. For example, a `Country` model might have many `Post` models through an intermediate `User` model. In this example, you could easily gather all blog posts for a given country. Let's look at the tables required to define this relationship:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Though `posts` does not contain a `country_id` column, the `hasManyThrough` relation provides access to a country's posts via `$country->posts`. To perform this query, Eloquent inspects the `country_id` on the intermediate `users` table. After finding the matching user IDs, they are used to query the `posts` table.

Now that we have examined the table structure for the relationship, let's define it on the `Country` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
     * Get all of the posts for the country.
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

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.

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, and the fifth argument is the local key:

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post', 'App\User',
            'country_id', 'user_id', 'id'
        );
    }
}
```

Polymorphic Relations

Table Structure

Polymorphic relations allow a model to belong to more than one other model on a single association. For example, imagine users of your application can "comment" both posts and videos. Using polymorphic relationships, you can use a single `comments` table for both of these scenarios. 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
```

Two important columns to note are the `commentable_id` and `commentable_type` columns on the `comments` table. The `commentable_id` column will contain the ID value of the post or video, while the `commentable_type` column will contain the class name of the owning model. The `commentable_type` column is how the ORM determines which "type" of owning model to return when accessing the `commentable` relation.

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
```

```
        return $this->morphMany('App\Comment', 'commentable');
    }
}
```

Retrieving Polymorphic Relations

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the comments for a post, we can simply use the `comments` dynamic property:

```
$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In our case, that is the `commentable` method on the `Comment` model. So, we will access that method as a dynamic property:

```
$comment = App\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 owns the comment.

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 example above where a `Comment` may belong to a `Post` or a `Video`, the default `commentable_type` would be either `App\Post` or `App\Video`, respectively. However, you may wish to decouple your database from your application's internal structure. In that case, you may define a relationship "morph map" to instruct Eloquent to use a custom name for each model instead of the class name:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'posts' => 'App\Post',
    'videos' => 'App\Video',
]);
```

You may register the `morphMap` in the `boot` function of your `AppServiceProvider` or create a separate service provider if you wish.

Many To Many Polymorphic Relations

Table Structure

In addition to traditional polymorphic relations, you may also define "many-to-many" polymorphic relations. For example, a blog `Post` and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation allows you to have a single list of unique tags that are shared across blog posts and videos. First, let's examine the table structure:

```
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
```

Model Structure

Next, we're ready to define the relationships on the model. The `Post` and `Video` models will both have a `tags` method that calls the `morphToMany` method on the base Eloquent class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
```

```
    }
}
```

Defining The Inverse Of The Relationship

Next, on the `Tag` model, you should define a method for each of its related models. So, for this example, we will define a `posts` method and a `videos` method:

```
<?php

namespace App;

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('App\Post', 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', '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 can simply use the `tags` dynamic property:

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphedByMany`. In our case, that is the `posts` or `videos` methods on the `Tag` model. So, you will access those methods as dynamic properties:

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

Querying Relations

Since all types of Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing the relationship queries. 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 against your database.

For example, imagine a blog system in which a `User` model has many associated `Post` models:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

```
$user = App\User::find(1);

$user->posts()->where('active', 1)->get();
```

You are able to use any of the query builder 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.

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may simply 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:

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Dynamic properties are "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 accessing the records for a model, 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` method:

```
// Retrieve all posts that have at least one comment...
$posts = App\Post::has('comments')->get();
```

You may also specify an operator and count to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may also be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment and vote:

```
// Retrieve all posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to put "where" conditions on your `has` queries. These methods allow you to add customized constraints to a relationship constraint, such as checking the content of a comment:

```
// Retrieve all posts with at least one comment containing words like foo%
$posts = Post::whereHas('comments', function ($query) {
```

```
    $query->where('content', 'like', 'foo%');
})->get();
```

Querying Relationship Absence

When accessing the records for a model, 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` method:

```
$posts = App\Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` method to put "where" conditions on your `doesntHave` queries. This method allows you to add customized constraints to a relationship constraint, such as checking the content of a comment:

```
$posts = Post::whereDoesntHave('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Counting Related Models

If you want to count the number of results from a relationship without actually loading them you may use the `withCount` method, which will place a `{relation}_count` column on your resulting models. For example:

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

You may add the "counts" for multiple relations as well as add constraints to the queries:

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->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:

```
$posts = Post::withCount([
    'comments',
```

```
    'comments AS pending_comments' => function ($query) {
        $query->where('approved', false);
    }
])->get();

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

Eager Loading

When accessing Eloquent relationships as properties, the relationship data is "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 is related to `Author`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Now, let's retrieve all books and their authors:

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, this loop would run 26 queries: 1 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 2 queries. When querying, you may specify which relationships should be eager loaded using the `with` method:

```
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed:

```
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 in a single operation. To do so, just pass additional arguments to the `with` method:

```
$books = App\Book::with('author', 'publisher')->get();
```

Nested Eager Loading

To eager load nested 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 in one Eloquent statement:

```
$books = App\Book::with('author.contacts')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship, but also specify additional query constraints for the eager loading query. Here's an example:

```
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `first`. Of course, you may call other query builder methods to further customize the eager loading operation:

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

```
$books = App\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:

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

Inserting & Updating Related Models

The Save Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to insert a new `Comment` for a `Post` model. Instead of manually setting the `post_id` attribute on the `Comment`, you may insert the `Comment` directly from the relationship's `save` method:

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

Notice 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 = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

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. Again, the difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`:

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Before using the `create` method, be sure to review the documentation on attribute mass assignment.

Belongs To Relationships

When updating a `belongsTo` relationship, you may use the `associate` method. This method will set the foreign key on the child model:

```
$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();
```

When removing a `belongsTo` relationship, 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 a few additional helper methods to make working with related models more convenient. For example, let's imagine a user can have many roles and a role can have many users. To attach a role to a user by inserting a record in the intermediate table that joins the models, use the `attach` method:

```
$user = App\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]);
```

Of course, 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 remove 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 = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

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, 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 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]);
```

Saving Additional Data On A Pivot Table

When working with a many-to-many relationship, the `save` method accepts an array of additional intermediate table attributes as its second argument:

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

Updating A Record On A Pivot Table

If you need to update an existing row in your pivot table, you may use `updateExistingPivot` method. This method accepts the pivot record foreign key and an array of attributes to update:

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

Touching Parent Timestamps

When a model `belongsTo` or `belongsToMany` 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`. Eloquent makes it easy. Just add a `touches` property containing the names of the relationships to the child model:

```
<?php

namespace App;

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('App\Post');
```

```
    }
}
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated as well, making it more convenient to know when to invalidate a cache of the `Post` model:

```
$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

Eloquent: Collections

Introduction

All multi-result sets returned by Eloquent are instances of the `Illuminate\Database\Eloquent\Collection` object, including results retrieved via the `get` method or accessed via a relationship. The Eloquent collection object extends the Laravel base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models.

Of course, all collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
$users = App\User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, 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, let's remove all inactive models and gather the first name for each remaining user:

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})
```

```
->map(function ($user) {
    return $user->name;
});
```

> {note} While most Eloquent collection methods return a new instance of an Eloquent collection, the `pluck`, `keys`, `zip`, `collapse`, `flatten` and `flip` methods return a base collection instance. Likewise, if a `map` operation returns a collection that does not contain any Eloquent models, it will be automatically cast to a base collection.

Available Methods

The Base Collection

All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class:

all average avg chunk collapse combine contains containsStrict count diff diffKeys each every except filter first flatMap flatten flip forget forPage get groupBy has implode intersect isEmpty isNotEmpty keyBy keys last map mapWithKeys max median merge min mode nth only partition pipe pluck pop prepend pull push put random reduce reject reverse search shift shuffle slice sort sortBy sortByDesc splice split sum take tap toArray toJson transform union unique uniqueStrict values when where whereStrict whereIn whereInStrict whereNotIn whereNotInStrict zip

Custom Collections

If you need to use a custom `Collection` object with your own extension methods, you may override the `newCollection` method on your model:

```
<?php

namespace App;

use App\CustomCollection;
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 $models = [])
    {
        return new CustomCollection($models);
    }
}
```

Once you have defined a `newCollection` method, you will receive an instance of your custom collection anytime Eloquent returns a `Collection` instance of that model. If you would like to use a custom collection for every model in your application, you should override the `newCollection` method on a base model class that is extended by all of your models.

Eloquent: Mutators

Introduction

Accessors and mutators allow you to format 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.

In addition to custom accessors and mutators, Eloquent can also automatically cast date fields to Carbon instances or even cast text fields to JSON.

Accessors & Mutators

Defining An Accessor

To define an accessor, create a `getFooAttribute` method on your model where `Foo` 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;

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:

```
$user = App\User::find(1);

$firstName = $user->first_name;
```

Defining A Mutator

To define a mutator, define a `setFooAttribute` method on your model where `Foo` is the "studly" cased name of the column you wish to access. So, again, 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;

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. So, for example, if we attempt to set the `first_name` attribute to `Sally`:

```
$user = App\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.

Date Mutators

By default, Eloquent will convert the `created_at` and `updated_at` columns to instances of Carbon, which extends the PHP `DateTime` class to provide an assortment of helpful methods. You may customize which dates are automatically mutated, and even completely disable this mutation, by overriding the `$dates` property of your model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = [
        'created_at',
        'updated_at',
```

```
        'deleted_at'
    ];
}
```

When a column is considered a date, you may set its value to a UNIX timestamp, date string (`Y-m-d`), date-time string, and of course a `DateTime` / `Carbon` instance, and the date's value will automatically be correctly stored in your database:

```
$user = App\User::find(1);

$user->deleted_at = Carbon::now();

$user->save();
```

As noted above, when retrieving attributes that are listed in your `$dates` property, they will automatically be cast to Carbon instances, allowing you to use any of Carbon's methods on your attributes:

```
$user = App\User::find(1);

return $user->deleted_at->getTimestamp();
```

Date Formats

By default, timestamps are formatted as `'Y-m-d H:i:s'`. If you need to customize the timestamp format, 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;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Attribute Casting

The `$casts` property on your model provides a convenient method of converting attributes to common data types. The `$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: `integer`, `real`, `float`, `double`, `string`, `boolean`, `object`, `array`, `collection`, `date`, `datetime`, and `timestamp`.

For example, 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;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

Now 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\User::find(1);

if ($user->is_admin) {
    //
}
```

Array & JSON Casting

The `array` cast type 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;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast to native types.
     *
     * @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:

```
$user = App\User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

Eloquent: Serialization

Introduction

When building JSON APIs, 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 your serializations.

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:

```
$user = App\User::with('roles')->first();

return $user->toArray();
```

You may also convert entire collections of models to arrays:

```
$users = App\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:

```
$user = App\User::find(1);

return $user->toJson();
```

Alternatively, you may cast a model or collection to a string, which will automatically call the `toJson` method on the model or collection:

```
$user = App\User::find(1);

return (string) $user;
```

Since models and collections are converted to JSON when cast to a string, you can return Eloquent objects directly from your application's routes or controllers:

```
Route::get('users', function () {
    return App\User::all();
});
```

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:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

> {note} When hiding relationships, use the relationship's method name, not its dynamic property name.

Alternatively, you may use the `visible` property to define a white-list of attributes that should be included in your model's array and JSON representation. All other attributes will be hidden when the model is converted to an array or JSON:

```
<?php

namespace App;

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 for convenient method chaining:

```
return $user->makeVisible('attribute')->toArray();
```

Likewise, if you would like to make some typically visible attributes hidden on a given model instance, you may use the `makeHidden` method.

```
return $user->makeHidden('attribute')->toArray();
```

Appending Values To JSON

Occasionally, when casting models to an array 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;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the administrator flag for the user.
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

After creating the accessor, add the attribute name to the `appends` property on the model. Note that attribute names are typically referenced in "snake case", even though the accessor is defined using "camel case":

```
<?php

namespace App;

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.

Testing: Getting Started

- Introduction
- Environment
- Creating & Running Tests

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a `phpunit.xml` file is already setup 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. 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.

An `ExampleTest.php` file is provided in both the `Feature` and `Unit` test directories. After installing a new Laravel application, simply run `phpunit` on the command line to run your tests.

Environment

When running tests via `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 the `phpunit.xml` file, but make sure to clear your configuration cache using the `config:clear` Artisan command before running your tests!

Creating & Running Tests

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

```
// Create a test in the Feature directory...
php artisan make:test UserTest

// Create a test in the Unit directory...
php artisan make:test UserTest --unit
```

Once the test has been generated, you may define test methods as you normally would using PHPUnit. To run your tests, simply execute the `phpunit` command from your terminal:

```
<?php

namespace Tests\Unit;

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

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

{note} If you define your own `setUp` method within a test class, be sure to call `parent::setUp()`.

HTTP Tests

Introduction

Laravel provides a very fluent API for making HTTP requests to your application and examining the output. For example, take a look at the test defined below:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $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.

Session / Authentication

Laravel provides several helpers for working 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

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

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

```
<?php

use App\User;

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(User::class)->create();

        $response = $this->actingAs($user)
                         ->withSession(['foo' => 'bar'])
                         ->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')
```

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `json`, `get`, `post`, `put`, `patch`, and `delete` methods may be used to issue requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/user` and assert that the expected data was returned:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

{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.

Verifying Exact Match

If you would like to verify that the given array is an **exact** match for the JSON returned by the application, you should use the `assertExactJson` method:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
```

```
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

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 Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
```

```
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

Fake File Customization

When creating files using the `fake` method, you may specify the width, height, and size of the image in order to better test your 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);
```

Available Assertions

Laravel provides a variety of custom assertion methods for your PHPUnit tests. These assertions may be accessed on the response that is returned from the `json`, `get`, `post`, `put`, and `delete` test methods:

Method	Description
`$response->assertStatus($code);`	Assert that the response has a given code.
`$response->assertRedirect($uri);`	Assert that the response is a redirect to a given URI.
`$response->assertHeader($headerName, $value = null);`	Assert that the given header is present on the response.
`$response->assertCookie($cookieName, $value = null);`	Assert that the response contains the given cookie.

Method	Description
`$response->assertPlainCookie($cookieName, $value = null);`	Assert that the response contains the given cookie (unencrypted).
`$response->assertSessionHas($key, $value = null);`	Assert that the session contains the given piece of data.
`$response->assertSessionHasErrors(array $keys);`	Assert that the session contains an error for the given field.
`$response->assertSessionMissing($key);`	Assert that the session does not contain the given key.
`$response->assertJson(array $data);`	Assert that the response contains the given JSON data.
`$response->assertJsonFragment(array $data);`	Assert that the response contains the given JSON fragment.
`$response->assertJsonMissing(array $data);`	Assert that the response does not contain the given JSON fragment.
`$response->assertExactJson(array $data);`	Assert that the response contains an exact match of the given JSON data.

Method	Description
`$response->assertJsonStructure(array $structure);`	Assert that the response has a given JSON structure.
`$response->assertViewHas($key, $value = null);`	Assert that the response view was given a piece of data.

Browser Tests (Laravel Dusk)

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 machine. 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 laravel/dusk
```

Once Dusk is installed, you should register the `Laravel\Dusk\DuskServiceProvider` service provider. You should register the provider within the `register` method of your `AppServiceProvider` in order to limit the environments in which Dusk is available, since it exposes the ability to log in as other users:

```
use Laravel\Dusk\DuskServiceProvider;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    if ($this->app->environment('local', 'testing')) {
        $this->app->register(DuskServiceProvider::class);
    }
}
```

Next, run the `dusk:install` Artisan command:

```
php artisan dusk:install
```

A `Browser` directory will be created within your `tests` directory and will contain an example test. Next, set the `APP_URL` environment variable in your `.env` file. This value should match the URL you use to access your application in a browser.

To run your tests, use the `dusk` Artisan command. The `dusk` command accepts any argument that is also accepted by the `phpunit` command:

```
php artisan dusk
```

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 simply 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', 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
```

Running Tests

To run your browser tests, use the `dusk` Artisan command:

```
php artisan dusk
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, allowing you to only run the tests for a given group, etc:

```
php artisan dusk --group=foo
```

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:

```
/**
 * 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.

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, call the `browse` method:

```
<?php

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Chrome;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $user = factory(User::class)->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                    ->type('email', $user->email)
                    ->type('password', 'secret')
```

```
                    ->press('Login')
                    ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a callback. A browser instance will automatically be passed to this callback by Dusk and is the main object used to interact with and make assertions against your application.

> {tip} This test can be used to test the login screen generated by the `make:auth` Artisan command.

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 "ask" for more than one browser in the signature of the callback 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');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with the login screen during every test. The `loginAs` method accepts a user ID or user model instance:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home');
});
```

Interacting With Elements

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);
```

> {note} This method interacts 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.

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 selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

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 an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Using 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 field with the given `name` attribute. Finally, Dusk will attempt to find a `textarea` with the given `name` attribute.

You may "clear" the value of an input using the `clear` method:

```
$browser->clear('email');
```

Dropdowns

To select a value in a dropdown selection box, 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 parameter:

```
$browser->select('size');
```

Checkboxes

To "check" a checkbox field, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If an exact selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');

$browser->uncheck('terms');
```

Radio Buttons

To "select" a radio button option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If an exact selector match can't be found, Dusk will search for a radio with matching `name` and `value` attributes:

```
$browser->radio('version', 'php7');
```

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 an exact

selector match can't be found, Dusk will search for a file input with matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/me.png');
```

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 hold modifier keys 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, `otwell` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

You may even send a "hot key" to the primary CSS selector that contains your application:

```
$browser->keys('.app', ['{command}', 'j']);
```

> {tip} All modifier keys 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 selector:

```
$browser->click('.selector');
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given 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', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

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 callback given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

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 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 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 given selector is missing from the page:

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for a given selector and then interact with the element matching the selector. 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 in this case. All element operations performed within the given callback 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);
```

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 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.dataLoaded');

$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);
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the table below:

Assertion	Description
`$browser->assertTitle($title)`	Assert the page title matches the given text.
`$browser->assertTitleContains($title)`	Assert the page title contains the given text.
`$browser->assertPathIs('/home')`	Assert the current path matches the given path.
`$browser->assertPathIsNot('/home')`	Assert the current path does not match the given path.
`$browser->assertQueryStringHas($name, $value)`	Assert the given query string parameter is present and has a given value.

Assertion	Description
`$browser->assertQueryStringMissing($name)`	Assert the given query string parameter is missing.
`$browser->assertHasCookie($name)`	Assert the given cookie is present.
`$browser->assertCookieValue($name, $value)`	Assert a cookie has a given value.
`$browser->assertPlainCookieValue($name, $value)`	Assert an unencrypted cookie has a given value.
`$browser->assertSee($text)`	Assert the given text is present on the page.
`$browser->assertDontSee($text)`	Assert the given text is not present on the page.
`$browser->assertSeeIn($selector, $text)`	Assert the given text is present within the selector.
`$browser->assertDontSeeIn($selector, $text)`	Assert the given text is not present within the selector.
`$browser->assertSeeLink($linkText)`	Assert the given link is present on the page.
`$browser->assertDontSeeLink($linkText)`	Assert the given link is not present on the page.
`$browser->assertInputValue($field, $value)`	Assert the given input field has the given value.

Assertion	Description
`$browser->assertInputValueIsNot($field, $value)`	Assert the given input field does not have the given value.
`$browser->assertChecked($field)`	Assert the given checkbox is checked.
`$browser->assertNotChecked($field)`	Assert the given checkbox is not checked.
`$browser->assertRadioSelected($field, $value)`	Assert the given radio field is selected.
`$browser->assertRadioNotSelected($field, $value)`	Assert the given radio field is not selected.
`$browser->assertSelected($field, $value)`	Assert the given dropdown has the given value selected.
`$browser->assertNotSelected($field, $value)`	Assert the given dropdown does not have the given value selected.
`$browser->assertValue($selector, $value)`	Assert the element matching the given selector has the given value.
`$browser->assertVisible($selector)`	Assert the element matching the given selector is visible.

Assertion	Description
`$browser->assertMissing($selector)`	Assert the element matching the given selector is not visible.

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Pages allow you to define expressive actions that may then be performed on a given page using a single method. Pages also allow you to define short-cuts to common selectors for your application or a single page.

Generating Pages

To generate a page object, use the `dusk:page` Artisan command. All page objects will be placed in the `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. Completing this method is not necessary; 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 configured, 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 of pages allows you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
```

```
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Now, you may use this shorthand selector anywhere you would 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 to the page method:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

Continuous Integration

Travis CI

To run your Dusk tests on Travis CI, we will need to use the "sudo-enabled" Ubuntu 14.04 (Trusty) environment. 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:

```
sudo: required
dist: trusty

before_script:
    - export DISPLAY=:99.0
    - sh -e /etc/init.d/xvfb start
    - ./vendor/laravel/dusk/bin/chromedriver-linux &
    - cp .env.testing .env
```

```
    - php artisan serve &

script:
    - php artisan dusk
```

CircleCI

If you are using CircleCI to run your Dusk tests, you may use this configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
test:
    pre:
        - "./vendor/laravel/dusk/bin/chromedriver-linux":
            background: true
        - cp .env.testing .env
        - "php artisan serve":
            background: true

    override:
        - php artisan dusk
```

Database Testing

Introduction

Laravel provides a variety of helpful tools to make it easier to test your database driven applications. First, you may use the `assertDatabaseHas` helper to assert that data exists in the database matching a given set of criteria. For example, if you would like to verify that there is a record in the `users` table with the `email` value of `sally@example.com`, you can do the following:

```
public function testDatabase()
{
    // Make call to application...

    $this->assertDatabaseHas('users', [
        'email' => 'sally@example.com'
    ]);
}
```

Of course, the `assertDatabaseHas` method and other helpers like it are for convenience. You are free to use any of PHPUnit's built-in assertion methods to supplement your tests.

Resetting The Database After Each Test

It is often useful to reset your database after each test so that data from a previous test does not interfere with subsequent tests.

Using Migrations

One approach to resetting the database state is to rollback the database after each test and migrate it before the next test. Laravel provides a simple `DatabaseMigrations` trait that will automatically handle this for you. Simply use the trait on your test class and everything will be handled for you:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
```

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

        // ...
    }
}
```

Using Transactions

Another approach to resetting the database state is to wrap each test case in a database transaction. Again, Laravel provides a convenient `DatabaseTransactions` trait that will automatically handle this for you:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseTransactions;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}
```

> {note} By default, this trait will only wrap the default database connection in a transaction. If your application is using multiple database connections, you should define a `$connectionsToTransact` property on your test class. This property should be an array of connection names to execute the transactions on.

Writing 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 default set of attributes for each of your Eloquent models using model factories. To get started, take a look at the `database/factories/ModelFactory.php` file in your application. Out of the box, this file contains one factory definition:

```
$factory->define(App\User::class, function (Faker\Generator $faker) {
    static $password;

    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => $password ?: $password = bcrypt('secret'),
        'remember_token' => str_random(10),
    ];
});
```

Within the Closure, which serves as the factory definition, you may return the default test values of all attributes on the model. The Closure will receive an instance of the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

Of course, you are free to add your own additional factories to the `ModelFactory.php` file. You may also create additional factory files for each model for better organization. For example, you could create `UserFactory.php` and `CommentFactory.php` files within your `database/factories` directory. All of the files within the `factories` directory will automatically be loaded by Laravel.

Factory States

States allow you to define discrete modifications that can be applied to your model factories in any combination. For example, your `User` model might have a `delinquent` state that modifies one of its default attribute values. You may define your state transformations using the `state` method:

```
$factory->state(App\User::class, 'delinquent', function ($faker) {
    return [
        'account_status' => 'delinquent',
    ];
});
```

Using Factories

Creating Models

Once you have defined your factories, you may use the global `factory` function in your tests or seed files to generate model instances. So, let's take a look at a few examples of creating models. First, we'll use the `make` method to create models but not save them to the database:

```
public function testDatabase()
{
    $user = factory(App\User::class)->make();

    // Use model in tests...
}
```

You may also create a Collection of many models or create models of a given type:

```
// Create three App\User instances...
$users = factory(App\User::class, 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 should specify the name of each state you would like to apply:

```
$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->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 values will be replaced while the rest of the values remain set to their default values as specified by the factory:

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

Persisting Models

The `create` method not only creates the model instances but also saves them to the database using Eloquent's `save` method:

```
public function testDatabase()
{
    // Create a single App\User instance...
    $user = factory(App\User::class)->create();

    // Create three App\User instances...
    $users = factory(App\User::class, 3)->create();

    // Use model in tests...
}
```

You may override attributes on the model by passing an array to the `create` method:

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

Relationships

In this example, we'll attach a relation to some created models. When using the `create` method to create multiple models, an Eloquent collection instance is returned, allowing you to use any of the convenient functions provided by the collection, such as `each`:

```
$users = factory(App\User::class, 3)
           ->create()
           ->each(function ($u) {
                $u->posts()->save(factory(App\Post::class)->make());
            });
```

Relations & Attribute Closures

You may also attach relationships to models using Closure attributes in your factory definitions. For example, if you would like to create a new `User` instance when creating a `Post`, you may do the following:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        }
    ];
});
```

These Closures also receive the evaluated attribute array of the factory that defines them:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        },
        'user_type' => function (array $post) {
            return App\User::find($post['user_id'])->type;
        }
    ];
});
```

Mocking

- Introduction
- Bus Fake
- Event Fake
- Mail Fake
- Notification Fake
- Queue Fake
- Storage Fake
- Facades

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 helpers for mocking events, jobs, and 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. Of course, you are free to use Mockery or PHPUnit to create your own mocks or spies.

Bus Fake

As an alternative to mocking, you may use the `Bus` facade's `fake` method to prevent jobs from being dispatched. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Bus;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Bus::fake();

        // Perform order shipping...

        Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}
```

Event Fake

As an alternative to mocking, you may use the `Event` facade's `fake` method to prevent all event listeners from executing. You may then assert that events were dispatched and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
```

```
use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function testOrderShipping()
    {
        Event::fake();

        // Perform order shipping...

        Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {
            return $e->order->id === $order->id;
        });

        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}
```

Mail Fake

You may use the `Mail` facade's `fake` method to prevent mail from being sent. You may then assert that mailables were sent to users and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
```

```
{
    public function testOrderShipping()
    {
        Mail::fake();

        // Perform order shipping...

        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // Assert a message was sent to the given users...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
            return $mail->hasTo($user->email) &&
                   $mail->hasCc('...') &&
                   $mail->hasBcc('...');
        });

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

Notification Fake

You may use the `Notification` facade's `fake` method to prevent notifications from being sent. You may then assert that notifications were sent to users and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
```

```
        Notification::fake();

        // Perform order shipping...

        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // 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
        );
    }
}
```

Queue Fake

As an alternative to mocking, you may use the `Queue` facade's `fake` method to prevent jobs from being queued. You may then assert that jobs were pushed to the queue and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
```

```
    {
        Queue::fake();

        // Perform order shipping...

        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::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 `UploadedFile` class, greatly simplifies the testing of file uploads. For example:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
```

```
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

Facades

Unlike traditional static method calls, facades may be mocked. This provides a great advantage over traditional static methods and grants you the same testability you would have if you were using dependency injection. When testing, you may often want to mock a call to a Laravel facade 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
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

We can mock the call to the `Cache` facade by using the `shouldReceive` method, which will return an instance of a Mockery mock. Since facades are actually resolved and managed by the 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 Tests\TestCase;
use Illuminate\Support\Facades\Cache;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

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 helper methods such as `get` and `post` when running your test. Likewise, instead of mocking the `Config` facade, simply call the `Config::set` method in your tests.

Laravel Cashier

- Introduction
- Configuration
  - Stripe
  - Braintree
  - Currency Configuration
- Subscriptions
  - Creating Subscriptions
  - Checking Subscription Status
  - Changing Plans
  - Subscription Quantity
  - Subscription Taxes
  - Cancelling Subscriptions
  - Resuming Subscriptions
  - Updating Credit Cards
- Subscription Trials
  - With Credit Card Up Front

Introduction

Laravel Cashier provides an expressive, fluent interface to Stripe's and Braintree'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.

> {note} If you're only performing "one-off" charges and do not offer subscriptions, you should not use Cashier. Instead, use the Stripe and Braintree SDKs directly.

Configuration

Stripe

Composer

First, add the Cashier package for Stripe to your `composer.json` file and run the `composer update` command:

```
"laravel/cashier": "~7.0"
```

Service Provider

Next, register the `Laravel\Cashier\CashierServiceProvider` service provider in your `config/app.php` configuration file.

Database Migrations

Before using Cashier, we'll also need to prepare the database. We need to add several columns to your `users` table and create a new `subscriptions` table to hold all of our customer's subscriptions:

```
Schema::table('users', function ($table) {
    $table->string('stripe_id')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('stripe_id');
    $table->string('stripe_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

Once the migrations have been created, run the `migrate` Artisan command.

Billable Model

Next, add the `Billable` trait to your model definition. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons, and updating credit card information:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Keys

Finally, you should configure your Stripe key in your `services.php` configuration file. You can retrieve your Stripe API keys from the Stripe control panel:

```
'stripe' => [
    'model'  => App\User::class,
    'key' => env('STRIPE_KEY'),
    'secret' => env('STRIPE_SECRET'),
],
```

Braintree

Braintree Caveats

For many operations, the Stripe and Braintree implementations of Cashier function the same. Both services provide subscription billing with credit cards but Braintree also supports payments via PayPal. However, Braintree also lacks some features that are supported by Stripe. You should keep the following in mind when deciding to use Stripe or Braintree:

- Braintree supports PayPal while Stripe does not.
- Braintree does not support the `increment` and `decrement` methods on subscriptions. This is a Braintree limitation, not a Cashier limitation.
- Braintree does not support percentage based discounts. This is a Braintree limitation, not a Cashier limitation.

Composer

First, add the Cashier package for Braintree to your `composer.json` file and run the `composer update` command:

```
"laravel/cashier-braintree": "~2.0"
```

Service Provider

Next, register the `Laravel\Cashier\CashierServiceProvider` service provider in your `config/app.php` configuration file.

Plan Credit Coupon

Before using Cashier with Braintree, you will need to define a `plan-credit` discount in your Braintree control panel. This discount will be used to properly prorate subscriptions that change from yearly to monthly billing, or from monthly to yearly billing.

The discount amount configured in the Braintree control panel can be any value you wish, as Cashier will simply override the defined amount with our own custom amount each time we apply the coupon. This coupon is needed since Braintree does not natively support prorating subscriptions across subscription frequencies.

Database Migrations

Before using Cashier, we'll need to prepare the database. We need to add several columns to your `users` table and create a new `subscriptions` table to hold all of our customer's subscriptions:

```
Schema::table('users', function ($table) {
    $table->string('braintree_id')->nullable();
    $table->string('paypal_email')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('braintree_id');
    $table->string('braintree_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

Once the migrations have been created, simply run the `migrate` Artisan command.

Billable Model

Next, add the `Billable` trait to your model definition:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Keys

Next, You should configure the following options in your `services.php` file:

```
'braintree' => [
    'model'  => App\User::class,
    'environment' => env('BRAINTREE_ENV'),
    'merchant_id' => env('BRAINTREE_MERCHANT_ID'),
    'public_key' => env('BRAINTREE_PUBLIC_KEY'),
    'private_key' => env('BRAINTREE_PRIVATE_KEY'),
],
```

Then you should add the following Braintree SDK calls to your `AppServiceProvider` service provider's `boot` method:

```
\Braintree_Configuration::environment(config('services.braintree.environment'));
\Braintree_Configuration::merchantId(config('services.braintree.merchant_id'));
\Braintree_Configuration::publicKey(config('services.braintree.public_key'));
\Braintree_Configuration::privateKey(config('services.braintree.private_key'));
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by calling the `Cashier::useCurrency` method from within the `boot` method of one of your service providers. The `useCurrency` method accepts two string parameters: the currency and the currency's symbol:

```
use Laravel\Cashier\Cashier;

Cashier::useCurrency('eur', '€');
```

Subscriptions

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($stripeToken);
```

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 `main` or `primary`. The second argument is the specific Stripe / Braintree plan the user is subscribing to. This value should correspond to the plan's identifier in Stripe or Braintree.

The `create` method will begin the subscription as well as update your database with the customer ID and other relevant billing information.

Additional User Details

If you would like to specify additional customer details, you may do so by passing them as the second argument to the `create` method:

```
$user->newSubscription('main', 'monthly')->create($stripeToken, [
    'email' => $email,
]);
```

To learn more about the additional fields supported by Stripe or Braintree, check out Stripe's documentation on customer creation or the corresponding Braintree documentation.

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$user->newSubscription('main', 'monthly')
     ->withCoupon('code')
     ->create($stripeToken);
```

Checking Subscription Status

Once a user is subscribed to your application, you may easily check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the user has an active subscription, even if the subscription is currently within its trial period:

```
if ($user->subscribed('main')) {
    //
}
```

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:

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed('main')) {
        // 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 displaying a warning to the user that they are still on their trial period:

```
if ($user->subscription('main')->onTrial()) {
    //
```

```
}
```

The `subscribedToPlan` method may be used to determine if the user is subscribed to a given plan based on a given Stripe / Braintree plan ID. In this example, we will determine if the user's `main` subscription is actively subscribed to the `monthly` plan:

```
if ($user->subscribedToPlan('monthly', 'main')) {
    //
}
```

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('main')->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('main')->onGracePeriod()) {
    //
}
```

Changing Plans

After a user is subscribed to your application, they may occasionally want to change to a new subscription plan. To swap a user to a new subscription, pass the plan's identifier to the `swap` method:

```
$user = App\User::find(1);

$user->subscription('main')->swap('provider-plan-id');
```

If the user is on 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('main')
        ->skipTrial()
        ->swap('provider-plan-id');
```

Subscription Quantity

> {note} Subscription quantities are only supported by the Stripe edition of Cashier. Braintree does not have a feature that corresponds to Stripe's "quantity".

Sometimes subscriptions are affected by "quantity". For example, your application might charge $10 per month **per user** on an account. To easily increment or decrement your subscription quantity, use the `incrementQuantity` and `decrementQuantity` methods:

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('main')->updateQuantity(10);
```

For more information on subscription quantities, consult the Stripe documentation.

Subscription Taxes

To specify the tax percentage a user pays on a subscription, implement the `taxPercentage` method on your billable model, and return a numeric value between 0 and 100, with no more than 2 decimal places.

```
public function taxPercentage() {
    return 20;
}
```

The `taxPercentage` method enables you to apply a tax rate on a model-by-model basis, which may be helpful for a user base that spans multiple countries and tax rates.

> {note} The `taxPercentage` method only applies to subscription charges. If you use Cashier to make "one off" charges, you will need to manually specify the tax rate at that time.

Cancelling Subscriptions

To cancel a subscription, simply call the `cancel` method on the user's subscription:

```
$user->subscription('main')->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.

You may determine if a user has cancelled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

If you wish to cancel a subscription immediately, call the `cancelNow` method on the user's subscription:

```
$user->subscription('main')->cancelNow();
```

Resuming Subscriptions

If a user has cancelled their subscription and you wish to resume it, use the `resume` method. The user **must** still be on their grace period in order to resume a subscription:

```
$user->subscription('main')->resume();
```

If the user cancels a subscription and then resumes that subscription before the subscription has fully expired, they will not be billed immediately. Instead, their subscription will simply be re-activated, and they will be billed on the original billing cycle.

Updating Credit Cards

The `updateCard` method may be used to update a customer's credit card information. This method accepts a Stripe token and will assign the new credit card as the default billing source:

```
$user->updateCard($stripeToken);
```

Subscription Trials

With Credit Card Up Front

If you would like to offer trial periods to your customers while still collecting payment method information up front, You should use the `trialDays` method when creating your subscriptions:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
            ->trialDays(10)
            ->create($stripeToken);
```

This method will set the trial period ending date on the subscription record within the database, as well as instruct Stripe / Braintree 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 identical:

```
if ($user->onTrial('main')) {
    //
}

if ($user->subscription('main')->onTrial()) {
    //
}
```

Without Credit Card Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may simply set the `trial_ends_at` column on the user record to your desired trial ending date. This is typically done during user registration:

```
$user = User::create([
    // Populate other user properties...
    'trial_ends_at' => Carbon::now()->addDays(10),
]);
```

> {note} Be sure to add a date mutator for `trial_ends_at` to your model definition.

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the `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...
}
```

You may also use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not created an actual subscription yet:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($stripeToken);
```

Handling Stripe Webhooks

Both Stripe and Braintree can notify your application of a variety of events via webhooks. To handle Stripe webhooks, define a route that points to Cashier's webhook controller. This controller will handle all incoming webhook requests and dispatch them to the proper controller method:

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

> {note} Once you have registered your route, be sure to configure the webhook URL in your Stripe control panel settings.

By default, this controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Stripe settings); however, as we'll soon discover, you can extend this controller to handle any webhook event you like.

Webhooks & CSRF Protection

Since Stripe webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your `VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'stripe/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellation on failed charges, but if you have additional Stripe webhook events you would like to handle, simply extend the Webhook controller. Your method names should correspond to Cashier's expected convention, specifically, methods should be prefixed with `handle` and the "camel case" name of the Stripe webhook you wish to handle. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you should add a `handleInvoicePaymentSucceeded` method to the controller:

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Stripe webhook.
     *
     * @param  array  $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

Failed Subscriptions

What if a customer's credit card expires? No worries - Cashier includes a Webhook controller that can easily cancel the customer's subscription for you. As noted above, all you need to do is point a route to the controller:

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

That's it! Failed payments will be captured and handled by the controller. The controller will cancel the customer's subscription when Stripe determines the subscription has failed (normally after three failed payment attempts).

Handling Braintree Webhooks

Both Stripe and Braintree can notify your application of a variety of events via webhooks. To handle Braintree webhooks, define a route that points to Cashier's webhook controller. This controller will handle all incoming webhook requests and dispatch them to the proper controller method:

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

{note} Once you have registered your route, be sure to configure the webhook URL in your Braintree control panel settings.

By default, this controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Braintree settings); however, as we'll soon discover, you can extend this controller to handle any webhook event you like.

Webhooks & CSRF Protection

Since Braintree webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your `VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'braintree/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellation on failed charges, but if you have additional Braintree webhook events you would like to handle, simply extend the Webhook controller. Your method names should correspond to Cashier's expected convention, specifically, methods should be prefixed with `handle` and the "camel case" name of the Braintree webhook you wish to handle. For example, if you wish to handle the `dispute_opened` webhook, you should add a `handleDisputeOpened` method to the controller:

```
<?php
```

```
namespace App\Http\Controllers;

use Braintree\WebhookNotification;
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Braintree webhook.
     *
     * @param  WebhookNotification  $webhook
     * @return Response
     */
    public function handleDisputeOpened(WebhookNotification $notification)
    {
        // Handle The Event
    }
}
```

Failed Subscriptions

What if a customer's credit card expires? No worries - Cashier includes a Webhook controller that can easily cancel the customer's subscription for you. Just point a route to the controller:

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

That's it! Failed payments will be captured and handled by the controller. The controller will cancel the customer's subscription when Braintree determines the subscription has failed (normally after three failed payment attempts). Don't forget: you will need to configure the webhook URI in your Braintree control panel settings.

Single Charges

Simple Charge

> {note} When using Stripe, the `charge` method accepts the amount you would like to charge in the **lowest denominator of the currency used by your application**. However, when using Braintree, you should pass the full dollar amount to the `charge` method:

If you would like to make a "one off" charge against a subscribed customer's credit card, you may use the `charge` method on a billable model instance.

```
// Stripe Accepts Charges In Cents...
$user->charge(100);

// Braintree Accepts Charges In Dollars...
$user->charge(1);
```

The `charge` method accepts an array as its second argument, allowing you to pass any options you wish to the underlying Stripe / Braintree charge creation. Consult the Stripe or Braintree documentation regarding the options available to you when creating charges:

```
$user->charge(100, [
    'custom_option' => $value,
]);
```

The `charge` method will throw an exception if the charge fails. If the charge is successful, the full Stripe / Braintree response will be returned from the method:

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

Charge With Invoice

Sometimes you may need to make a one-time charge but also generate an invoice for the charge so that you may offer a PDF receipt to your customer. The `invoiceFor` method lets you do just that. For example, let's invoice the customer $5.00 for a "One Time Fee":

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);

// Braintree Accepts Charges In Dollars...
$user->invoiceFor('One Time Fee', 5);
```

The invoice will be charged immediately against the user's credit card. The `invoiceFor` method also accepts an array as its third argument, allowing you to pass any options you wish to the underlying Stripe / Braintree charge creation:

```
$user->invoiceFor('One Time Fee', 500, [
    'custom-option' => $value,
]);
```

{note} The `invoiceFor` method will create a Stripe invoice which will retry failed billing attempts. If you do not want invoices to retry failed charges, you will need to close them using the Stripe API after the first failed charge.

Invoices

You may easily retrieve an array of a billable model's invoices using the `invoices` method:

```
$invoices = $user->invoices();

// Include pending invoices in the results...
$invoices = $user->invoicesIncludingPending();
```

When listing the invoices for the customer, you may use the invoice's helper methods to display the relevant invoice information. For example, you may wish to list every invoice in a table, allowing the user to easily download any of them:

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>
```

Generating Invoice PDFs

Before generating invoice PDFs, you need to install the `dompdf` PHP library:

```
composer require dompdf/dompdf
```

Then, from within a route or controller, use the `downloadInvoice` method to generate a PDF download of the invoice. This method will automatically generate the proper HTTP response to send the download to the browser:

```
use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor'  => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

Envoy Task Runner

Introduction

Laravel Envoy provides a clean, minimal syntax for defining 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.

Installation

First, install Envoy using the Composer `global require` command:

```
composer global require "laravel/envoy=~1.0"
```

Since global Composer libraries can sometimes cause package version conflicts, you may wish to consider using `cgr`, which is a drop-in replacement for the `composer global require` command. The `cgr` library's installation instructions can be found on GitHub.

> {note} Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `envoy` executable is found when running the `envoy` command in your terminal.

Updating Envoy

You may also use Composer to keep your Envoy installation up to date. Issuing the `composer global update` command will update all of your globally installed Composer packages:

```
composer global update
```

Writing Tasks

All of your Envoy tasks should be defined in an `Envoy.blade.php` file in the root of your project. Here's an example to get you started:

```
@servers(['web' => ['user@192.168.1.1']])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

As you can see, an array of `@servers` is defined at the top of the file, allowing you to reference these servers in the `on` option of your task declarations. Within your `@task` declarations, you should place the Bash code that should run on your server when the task is executed.

You can force a script to run locally by specifying the server's IP address as `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Setup

Sometimes, you may need to execute some PHP code before executing your Envoy tasks. You may use the `@setup` directive to declare variables and do other general PHP work before any of your other tasks are executed:

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@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('foo')
    # ...
@endtask
```

Variables

If needed, you may pass option values into Envoy tasks using the command line:

```
envoy run deploy --branch=master
```

You may access the options in your tasks via Blade's "echo" syntax. Of course, you may also use `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' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate
@endtask
```

Stories

Stories group a set of tasks under a single, convenient name, allowing you to group small, focused tasks into large tasks. For instance, a `deploy` story may run the `git` and `composer` tasks by listing the task names within its definition:

```
@servers(['web' => '192.168.1.1'])

@story('deploy')
    git
    composer
@endstory

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask
```

Once the story has been written, you may run it just like a typical task:

```
envoy run deploy
```

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, 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 site
    git pull origin {{ $branch }}
    php artisan migrate
@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 site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Running Tasks

To run a task or story that is defined in your `Envoy.blade.php` file, execute Envoy's `run` command, passing the name of the task or story you would like to execute. Envoy will run the task and display the output from the servers as the task is running:

```
envoy run task
```

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 site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Notifications

Slack

Envoy also supports sending notifications to Slack after each task is executed. The `@slack` directive accepts a Slack hook URL and a channel 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 into the `@slack` directive:

```
@finished
    @slack('webhook-url', '#bots')
@endfinished
```

You may provide one of the following as the channel argument:

- To send the notification to a channel: `#channel`
- To send the notification to a user: `@user`

API Authentication (Passport)

- Introduction
- Installation
  - Frontend Quickstart
  - Deploying Passport
- Configuration
  - Token Lifetimes
- Issuing Access Tokens
  - Managing Clients
  - Requesting Tokens
  - Refreshing Tokens
- Password Grant Tokens
  - Creating A Password Grant Client
  - Requesting Tokens
  - Requesting All Scopes
- 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

Introduction

Laravel already makes it easy to perform authentication via traditional login forms, but what about APIs? APIs typically use tokens to authenticate users and do not maintain session state between requests. Laravel makes API authentication a breeze using Laravel Passport, which 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 Alex Bilbie.

> {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.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Next, register the Passport service provider in the `providers` array of your `config/app.php` configuration file:

```
Laravel\Passport\PassportServiceProvider::class,
```

The Passport service provider registers its own database migration directory with the framework, so you should migrate your database after registering the provider. The Passport migrations will create the tables your application needs to store clients and access tokens:

```
php artisan migrate
```

> {note} If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `AppServiceProvider`. You may export the default migrations using `php artisan vendor:publish --tag=passport-migrations`.

Next, you should run the `passport:install` 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
```

After running this command, add the `Laravel\Passport\HasApiTokens` trait to your `App\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;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

Next, you should call the `Passport::routes` method within the `boot` method of your `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 Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
```

```
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, in your `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',
    ],
],
```

Frontend Quickstart

> {note} In order to use the Passport Vue components, you must be using the Vue JavaScript framework. These components also use the Bootstrap CSS framework. However, even if you are not using these tools, the components serve as a valuable reference for your own frontend implementation.

Passport ships with a JSON API that you may use to allow your users to create clients and personal access tokens. However, it can be time consuming to code a frontend to interact with these APIs. So, Passport also includes pre-built Vue components you may use as an example implementation or starting point for your own implementation.

To publish the Passport Vue components, use the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-components
```

The published components will be placed in your `resources/assets/js/components` directory. Once the components have been published, you should register them in your `resources/assets/js/app.js` file:

```
Vue.component(
```

```
    'passport-clients',
    require('./components/passport/Clients.vue')
);

Vue.component(
    'passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
    'passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue')
);
```

After registering the components, make sure to run `npm run dev` to recompile your assets. Once you have recompiled your assets, you may drop the components into one of your application's templates to get started creating clients and personal access tokens:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Deploying Passport

When deploying Passport to your production 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 token. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

Configuration

Token Lifetimes

By default, Passport issues long-lived access tokens that never need to be refreshed. If you would like to configure a shorter token lifetime, you may use the `tokensExpireIn` and `refreshTokensExpireIn` methods. These methods should be called from the `boot` method of your `AuthServiceProvider`:

```
use Carbon\Carbon;

/**
 * Register any authentication / authorization services.
 *
```

```
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(Carbon::now()->addDays(15));

    Passport::refreshTokensExpireIn(Carbon::now()->addDays(30));
}
```

Issuing Access Tokens

Using OAuth2 with 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
```

JSON API

Since your 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.

> {tip} If you don't want to implement the entire client management frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`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:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://your-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 `resources/views/vendor/passport`:

```
php artisan vendor:publish --tag=passport-views
```

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 then 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. In this example, we'll use the Guzzle HTTP library to make the `POST` request:

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);

    return json_decode((string) $response->getBody(), true);
});
```

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.

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. In this example, we'll use the Guzzle HTTP library to refresh the token:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

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.

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 e-mail 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` 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:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

> {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, 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` grant:

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
]);
```

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 your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Once a 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:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-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. To retrieve a token, make a request to the `oauth/token` endpoint:

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

echo json_decode((string) $response->getBody(), true);
```

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.

> {note} Personal access tokens are always long-lived. Their lifetime is not modified when using the `tokensExpireIn` or `refreshTokensExpireIn` methods.

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 using the `passport:client` 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
```

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 `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:

```
$user = App\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.

> {tip} If you don't want to implement the personal access token frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`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 token so that they may edit or delete 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 delete 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 require a valid access token:

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

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:

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);
```

Token Scopes

Defining 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.

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your `AuthServiceProvider`. 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:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

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://your-app.com/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `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('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('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 `User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('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:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

This Passport 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. Now, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/user')
    .then(response => {
        console.log(response.data);
    });
```

When using this method of authentication, Axios will automatically send the `X-CSRF-TOKEN` header. In addition, the default Laravel JavaScript scaffolding instructs Axios to send the `X-Requested-With` header:

```
window.axios.defaults.headers.common = {
    'X-Requested-With': 'XMLHttpRequest',
};
```

> {note} If you are using a different JavaScript framework, you should make sure it is configured to send the `X-CSRF-TOKEN` and `X-Requested-With` headers with every outgoing request.

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. You may attach listeners to these events in your application's `EventServiceProvider`:

“‘php /** * 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:

```
public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```

Laravel Scout

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 the Scout via the Composer package manager:

```
composer require laravel/scout
```

Next, you should add the `ScoutServiceProvider` to the `providers` array of your `config/app.php` configuration file:

```
Laravel\Scout\ScoutServiceProvider::class,
```

After registering the Scout service provider, you should publish the Scout configuration using the `vendor:publish` Artisan command. This command will publish the `scout.php` configuration file to your `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 to keep the model in sync with your search driver:

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;
}
```

Queueing

While not strictly required to use Scout, you should strongly consider configuring a queue driver before using the library. Running a queue worker will allow Scout to queue all operations that sync your model information to your search indexes, providing much better response times for your application's web interface.

Once you have configured a queue driver, set the value of the `queue` option in your `config/scout.php` configuration file to `true`:

```
'queue' => true,
```

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
```

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;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
```

```
     * Get the index name for 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;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize array...

        return $array;
    }
}
```

Indexing

Batch Import

If you are installing Scout into an existing project, you may already have database records you need to import into your search driver. Scout provides an `import` Artisan command that you may use to import all of your existing records into your search indexes:

```
php artisan scout:import "App\Post"
```

Adding Records

Once you have added the `Laravel\Scout\Searchable` trait to a model, all you need to do is `save` 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:

```
$order = new App\Order;

// ...

$order->save();
```

Adding 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 an 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 added in the background by your queue workers:

```
// Adding via Eloquent query...
App\Order::where('price', '>', 100)->searchable();

// You may also add records via relationships...
$user->orders()->searchable();

// You may also add records via collections...
$orders->searchable();
```

The `searchable` method can be considered an "upsert" operation. In other words, if the model record is already in your index, it will be updated. If it does not exist in the search index, it will be added to the index.

Updating Records

To update a searchable model, you only need to update the model instance's properties and `save` the model to your database. Scout will automatically persist the changes to your search index:

```
$order = App\Order::find(1);

// Update the order...

$order->save();
```

You may also use the `searchable` method on an Eloquent query to update a collection of models. If the models do not exist in your search index, they will be created:

```
// Updating via Eloquent query...
App\Order::where('price', '>', 100)->searchable();

// You may also update via relationships...
$user->orders()->searchable();

// You may also update via collections...
$orders->searchable();
```

Removing Records

To remove a record from your index, simply `delete` the model from the database. This form of removal is even compatible with soft deleted models:

```
$order = App\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 or collection:

```
// Removing via Eloquent query...
App\Order::where('price', '>', 100)->unsearchable();

// You may also remove via relationships...
$user->orders()->unsearchable();

// You may also remove via collections...
$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 callback which will be immediately executed. Any model operations that occur within the callback will not be synced to the model's index:

```
App\Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

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:

```
$orders = App\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 Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->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 a tenant ID. Since a search index is not a relational database, more advanced "where" clauses are not currently supported:

```
$orders = App\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 a `Paginator` instance just as if you had paginated a traditional Eloquent query:

```
$orders = App\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 = App\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() }}
```

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 five 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, $page);
abstract public function map($results, $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. You should call the `extend` method from the `boot` method of your `AppServiceProvider` or any other service provider used by your application. For example, if you have written a `MySqlSearchEngine`, you may register it like so:

```
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 `config/scout.php` configuration file:

```
'driver' => 'mysql',
```