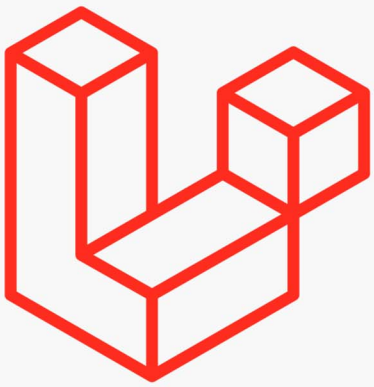


Laravel advanced book

auto generated from the official docs

Table of Contents

Welcome

This book is automatically created from the markdown files of the official Laravel Documentation (laravel/docs).
This second book gives you an advanced overview of what Laravel can do with it's advanced toolset.

The first book laravel-starter-book.pdf is the more basic part of the documentation and is a good starter guide.

Disclaimer: none of the content of this book is authored by Lloadout, all content is auto-generated from the official docs.

Artisan Console

Introduction

Artisan is the command line interface included with Laravel. Artisan exists at the root of your application as the `artisan` script and provides a number of helpful commands that can assist you while you build your application. To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

Every command also includes a "help" screen which displays and describes the command's available arguments and options. To view a help screen, precede the name of the command with `help`:

```
php artisan help migrate
```

Laravel Sail

If you are using Laravel Sail as your local development environment, remember to use the `sail` command line to invoke Artisan commands. Sail will execute your Artisan commands within your application's Docker containers:

```
./sail artisan list
```

Tinker (REPL)

Laravel Tinker is a powerful REPL for the Laravel framework, powered by the PsySH package.

Installation

All Laravel applications include Tinker by default. However, you may install Tinker using Composer if you have previously removed it from your application:

```
composer require laravel/tinker
```

{tip} Looking for a graphical UI for interacting with your Laravel application? Check out Tinkerwell!

Usage

Tinker allows you to interact with your entire Laravel application on the command line, including your Eloquent models, jobs, events, and more. To enter the Tinker environment, run the `tinker` Artisan command:

```
php artisan tinker
```

You can publish Tinker's configuration file using the `vendor:publish` command:

```
php artisan vendor:publish --
provider="Laravel\Tinker\TinkerServiceProvider"
```

{note} The `dispatch` helper function and `dispatch` method on the `Dispatchable` class depends on garbage collection to place the job on the queue. Therefore, when using tinker, you should use `Bus::dispatch` or `Queue::push` to dispatch jobs.

Command Allow List

Tinker utilizes an "allow" list to determine which Artisan commands are allowed to be run within its shell. By default, you may run the `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize`, and `up` commands. If you would like to allow more commands you may add them to the `commands` array in your `tinker.php` configuration file:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

Classes That Should Not Be Aliased

Typically, Tinker automatically aliases classes as you interact with them in Tinker. However, you may wish to never alias some classes. You may accomplish this by listing the classes in the `dont_alias` array of your `tinker.php` configuration file:

```
'dont_alias' => [
    App\Models\User::class,
],
```

Writing Commands

In addition to the commands provided with Artisan, you may build your own custom commands. Commands are typically stored in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be loaded by Composer.

Generating Commands

To create a new command, you may use the `make:command` Artisan command. This command will create a new command class in the `app/Console/Commands` directory. Don't worry if this directory does not exist in your application - it will be created the first time you run the `make:command` Artisan command:

```
php artisan make:command SendEmails
```

Command Structure

After generating your command, you should define appropriate values for the `signature` and `description` properties of the class. These properties will be used when displaying your command on the `list` screen. The `signature` property also allows you to define your command's input expectations. The `handle` method will be called when your command is executed. You may place your command logic in this method.

Let's take a look at an example command. Note that we are able to request any dependencies we need via the command's `handle` method. The Laravel service container will automatically inject all dependencies that are type-hinted in this method's signature:

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use App\Support\DripEmailer;
use Illuminate\Console\Command;
```

```
class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send a marketing email to a user';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @param  \App\Support\DripEmailer  $drip
     * @return mixed
     */
    public function handle(DripEmailer $drip)
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

{tip} For greater code reuse, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks. In the example above, note that we inject a service class to do the "heavy lifting" of sending the e-mails.

Closure Commands

Closure based commands provide an alternative to defining console commands as classes. In the same way that route closures are an alternative to controllers, think of command closures as an alternative to command classes. Within the `commands` method of your `app/Console/Kernel.php` file, Laravel loads the `routes/console.php` file:

```
/**
 * Register the closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}
```

Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application. Within this file, you may define all of your closure based console commands using the `Artisan::command` method. The `command` method accepts two arguments: the command signature and a closure which receives the command's arguments and options:

```
Artisan::command('mail:send {user}', function ($user) {
    $this->info("Sending email to: {$user}!");
});
```

The closure is bound to the underlying command instance, so you have full access to all of the helper methods you would typically be able to access on a full command class.

Type-Hinting Dependencies

In addition to receiving your command's arguments and options, command closures may also type-hint additional dependencies that you would like resolved out of the service container:

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, $user)
{
    $drip->send(User::find($user));
});
```

Closure Command Descriptions

When defining a closure based command, you may use the `purpose` method to add a description to the command. This description will be displayed when you run the `php artisan list` or `php artisan help` commands:

```
Artisan::command('mail:send {user}', function ($user) {
    // ...
})->purpose('Send a marketing email to a user');
```

Defining Input Expectations

When writing console commands, it is common to gather input from the user through arguments or options. Laravel makes it very convenient to define the input you expect from the user using the `signature` property on your commands. The `signature` property allows you to define the name, arguments, and options for the command in a single, expressive, route-like syntax.

Arguments

All user supplied arguments and options are wrapped in curly braces. In the following example, the command defines one required argument: `user`:

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

You may also make arguments optional or define default values for arguments:

```
// Optional argument...
mail:send {user?}

// Optional argument with default value...
mail:send {user=foo}
```

Options

Options, like arguments, are another form of user input. Options are prefixed by two hyphens (`--`) when they are provided via the command line. There are two types of options: those that receive a value and those that don't. Options that don't receive a value serve as a boolean "switch". Let's take a look at an example of this type of option:

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

In this example, the `--queue` switch may be specified when calling the Artisan command. If the `--queue` switch is passed, the value of the option will be `true`. Otherwise, the value will be `false`:

```
php artisan mail:send 1 --queue
```

Options With Values

Next, let's take a look at an option that expects a value. If the user must specify a value for an option, you should suffix the option name with a `=` sign:

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

In this example, the user may pass a value for the option like so. If the option is not specified when invoking the command, its value will be `null`:

```
php artisan mail:send 1 --queue=default
```

You may assign default values to options by specifying the default value after the option name. If no option value is passed by the user, the default value will be used:

```
mail:send {user} {--queue=default}
```

Option Shortcuts

To assign a shortcut when defining an option, you may specify it before the option name and use the `|` character as a delimiter to separate the shortcut from the full option name:

```
mail:send {user} {--Q|queue}
```

Input Arrays

If you would like to define arguments or options to expect multiple input values, you may use the `*` character. First, let's take a look at an example that specifies such an argument:

```
mail:send {user*}
```

When calling this method, the `user` arguments may be passed in order to the command line. For example, the following command will set the value of `user` to an array with `foo` and `bar` as its values:

```
php artisan mail:send foo bar
```

This `*` character can be combined with an optional argument definition to allow zero or more instances of an argument:

```
mail:send {user?*}
```

Option Arrays

When defining an option that expects multiple input values, each option value passed to the command should be prefixed with the option name:

```
mail:send {user} {--id=*}

php artisan mail:send --id=1 --id=2
```

Input Descriptions

You may assign descriptions to input arguments and options by separating the argument name from the description using a colon. If you need a little extra room to define your command, feel free to spread the definition across multiple lines:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send
                        {user : The ID of the user}
                        {--queue : Whether the job should be queued}';
```

Command I/O

Retrieving Input

While your command is executing, you will likely need to access the values for the arguments and options accepted by your command. To do so, you may use the `argument` and `option` methods. If an argument or option does not exist, `null` will be returned:

```
/**
 * Execute the console command.
 *
 * @return int
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

If you need to retrieve all of the arguments as an `array`, call the `arguments` method:

```
$arguments = $this->arguments();
```

Options may be retrieved just as easily as arguments using the `option` method. To retrieve all of the options as an array, call the `options` method:

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options as an array...
$options = $this->options();
```

Prompting For Input

In addition to displaying output, you may also ask the user to provide input during the execution of your command. The `ask` method will prompt the user with the given question, accept their input, and then return the user's input back to your command:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $name = $this->ask('What is your name?');
}
```

The `secret` method is similar to `ask`, but the user's input will not be visible to them as they type in the console. This method is useful when asking for sensitive information such as passwords:

```
$password = $this->secret('What is the password?');
```

Asking For Confirmation

If you need to ask the user for a simple "yes or no" confirmation, you may use the `confirm` method. By default, this method will return `false`. However, if the user enters `y` or `yes` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    //
}
```

If necessary, you may specify that the confirmation prompt should return `true` by default by passing `true` as the second argument to the `confirm` method:

```
if ($this->confirm('Do you wish to continue?', true)) {
    //
}
```

Auto-Completion

The `anticipate` method can be used to provide auto-completion for possible choices. The user can still provide any answer, regardless of the auto-completion hints:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Alternatively, you may pass a closure as the second argument to the `anticipate` method. The closure will be called each time the user types an input character. The closure should accept a string parameter containing the user's input so far, and return an array of options for auto-completion:

```
$name = $this->anticipate('What is your address?', function ($input) {
    // Return auto-completion options...
});
```

Multiple Choice Questions

If you need to give the user a predefined set of choices when asking a question, you may use the `choice` method. You may set the array index of the default value to be returned if no option is chosen by passing the index as the third argument to the method:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex
);
```

In addition, the `choice` method accepts optional fourth and fifth arguments for determining the maximum number of attempts to select a valid response and whether multiple selections are permitted:

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex,
    $maxAttempts = null,
    $allowMultipleSelections = false
);
```

Writing Output

To send output to the console, you may use the `line`, `info`, `comment`, `question`, `warn`, and `error` methods. Each of these methods will use appropriate ANSI colors for their purpose. For example, let's display some general information to the user. Typically, the `info` method will display in the console as green colored text:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    // ...

    $this->info('The command was successful!');
}
```

To display an error message, use the `error` method. Error message text is typically displayed in red:

```
$this->error('Something went wrong!');
```

You may use the `line` method to display plain, uncolored text:

```
$this->line('Display this on the screen');
```

You may use the `newLine` method to display a blank line:

```
// Write a single blank line...
$this->newLine();

// Write three blank lines...
$this->newLine(3);
```

Tables

The `table` method makes it easy to correctly format multiple rows / columns of data. All you need to do is provide the column names and the data for the table and Laravel will automatically calculate the appropriate width and height of the table for you:

```
use App\Models\User;

$this->table(
    ['Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

Progress Bars

For long running tasks, it can be helpful to show a progress bar that informs users how complete the task is. Using the `withProgressBar` method, Laravel will display a progress bar and advance its progress for each iteration over a given iterable value:

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function ($user) {
    $this->performTask($user);
});
```

Sometimes, you may need more manual control over how a progress bar is advanced. First, define the total number of steps the process will iterate through. Then, advance the progress bar after processing each item:

```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

{tip} For more advanced options, check out the Symfony Progress Bar component documentation.

Registering Commands

All of your console commands are registered within your application's `App\Console\Kernel` class, which is your application's "console kernel". Within the `commands` method of this class, you will see a call to the kernel's `load` method. The `load` method will scan the `app/Console/Commands` directory and automatically register each command it contains with Artisan. You are even free to make additional calls to the `load` method to scan other directories for Artisan commands:

```
/**
 * Register the commands for the application.
 *
 * @return void
 */
protected function commands()
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/../Domain/Orders/Commands');

    // ...
}
```

If necessary, you may manually register commands by adding the command's class name to a `$commands` property within your `App\Console\Kernel` class. If this property is not already defined on your kernel, you should define it manually. When Artisan boots, all the commands listed in this property will be resolved by the service container and registered with Artisan:

```
protected $commands = [
    Commands\SendEmails::class
];
```

Programmatically Executing Commands

Sometimes you may wish to execute an Artisan command outside of the CLI. For example, you may wish to execute an Artisan command from a route or controller. You may use the `call` method on the `Artisan` facade to accomplish this. The `call` method accepts either the command's signature name or class name as its first argument, and an array of command parameters as the second argument. The exit code will be returned:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    //
});
```

Alternatively, you may pass the entire Artisan command to the `call` method as a string:

```
Artisan::call('mail:send 1 --queue=default');
```

Passing Array Values

If your command defines an option that accepts an array, you may pass an array of values to that option:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

Passing Boolean Values

If you need to specify the value of an option that does not accept string values, such as the `--force` flag on the `migrate:refresh` command, you should pass `true` or `false` as the value of the option:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Queueing Artisan Commands

Using the `queue` method on the `Artisan` facade, you may even queue Artisan commands so they are processed in the background by your queue workers. Before using this method, make sure you have configured your queue and are running a queue listener:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    //
});
```

Using the `onConnection` and `onQueue` methods, you may specify the connection or queue the Artisan command should be dispatched to:

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Calling Commands From Other Commands

Sometimes you may wish to call other commands from an existing Artisan command. You may do so using the `call` method. This `call` method accepts the command name and an array of command arguments / options:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
}
```

If you would like to call another console command and suppress all of its output, you may use the `callSilently` method. The `callSilently` method has the same signature as the `call` method:

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Signal Handling

The Symfony Console component, which powers the Artisan console, allows you to indicate which process signals (if any) your command handles. For example, you may indicate that your command handles the `SIGINT` and `SIGTERM` signals.

To get started, you should implement the `Symfony\Component\Console\Command\SignalableCommandInterface` interface on your Artisan command class. This interface requires you to define two methods: `getSubscribedSignals` and `handleSignal`:

```php
<?php

use Symfony\Component\Console\Command\SignalableCommandInterface;

class StartServer extends Command implements SignalableCommandInterface
{
    // ...

    /**
     * Get the list of signals handled by the command.
     *
     * @return array
     */
    public function getSubscribedSignals(): array
    {
        return [SIGINT, SIGTERM];
    }

    /**
     * Handle an incoming signal.
     *
     * @param  int  $signal
     * @return void
     */
    public function handleSignal(int $signal): void
    {
        if ($signal === SIGINT) {
            $this->stopServer();

            return;
        }
    }
}
```

As you might expect, the `getSubscribedSignals` method should return an array of the signals that your command can handle, while the `handleSignal` method receives the signal and can respond accordingly.

Stub Customization

The Artisan console's `make` commands are used to create a variety of classes, such as controllers, jobs, migrations, and tests. These classes are generated using "stub" files that are populated with values based on your input. However, you may want to make small changes to files generated by Artisan. To accomplish this, you may use the `stub:publish` command to publish the most common stubs to your application so that you can customize them:

```
php artisan stub:publish
```

The published stubs will be located within a `stubs` directory in the root of your application. Any changes you make to these stubs will be reflected when you generate their corresponding classes using Artisan's `make` commands.

Events

Artisan dispatches three events when running commands: `Illuminate\Console\Events\ArtisanStarting`, `Illuminate\Console\Events\CommandStarting`, and `Illuminate\Console\Events\CommandFinished`. The `ArtisanStarting` event is dispatched immediately when Artisan starts running. Next, the `CommandStarting` event is dispatched immediately before a command runs. Finally, the `CommandFinished` event is dispatched once a command finishes executing.

Laravel Cashier (Stripe)

Introduction

Laravel Cashier provides an expressive, fluent interface to Stripe's subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading writing. In addition to basic subscription management, Cashier can handle coupons, swapping subscription, subscription "quantities", cancellation grace periods, and even generate invoice PDFs.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review the upgrade guide.

{note} To prevent breaking changes, Cashier uses a fixed Stripe API version. Cashier 13 utilizes Stripe API version `2020-08-27`. The Stripe API version will be updated on minor releases in order to make use of new Stripe features and improvements.

Installation

First, install the Cashier package for Stripe using the Composer package manager:

```
composer require laravel/cashier
```

{note} To ensure Cashier properly handles all Stripe events, remember to set up Cashier's webhook handling.

Database Migrations

Cashier's service provider registers its own database migration directory, so remember to migrate your database after installing the package. The Cashier migrations will add several columns to your `users` table as well as create a new `subscriptions` table to hold all of your customer's subscriptions:

```
php artisan migrate
```

If you need to overwrite the migrations that ship with Cashier, you can publish them using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

If you would like to prevent Cashier's migrations from running entirely, you may use the `ignoreMigrations` method provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Cashier\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::ignoreMigrations();
}
```

{note} Stripe recommends that any column used for storing Stripe identifiers should be case-sensitive. Therefore, you should ensure the column collation for the `stripe_id` column is set to `utf8_bin` when using MySQL. More information regarding this can be found in the Stripe documentation.

Configuration

Billable Model

Before using Cashier, add the `Billable` trait to your billable model definition. Typically, this will be the `App\Models\User` model. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons, and updating payment method information:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

Cashier assumes your billable model will be the `App\Models\User` class that ships with Laravel. If you wish to change this you may specify a different model via the `useCustomerModel` method. This method should typically be called in the `boot` method of your `AppServiceProvider` class:

```
use App\Models\Cashier\User;
use Laravel\Cashier\Cashier;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Cashier::useCustomerModel(User::class);
}
```

{note} If you're using a model other than Laravel's supplied `App\Models\User` model, you'll need to publish and alter the Cashier migrations provided to match your alternative model's table name.

API Keys

Next, you should configure your Stripe API keys in your application's `.env` file. You can retrieve your Stripe API keys from the Stripe control panel:

```
STRIPE_KEY=your-stripe-key
STRIPE_SECRET=your-stripe-secret
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by setting the `CASHIER_CURRENCY` environment variable within your application's `.env` file:

```
CASHIER_CURRENCY=eur
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's `NumberFormatter` class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```

{note} In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Tax Configuration

Thanks to Stripe Tax, it's possible to automatically calculate taxes for all invoices generated by Stripe. You can enable automatic tax calculation by invoking the `calculateTaxes` method in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use Laravel\Cashier\Cashier;

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

Once tax calculation has been enabled, any new subscriptions and any one-off invoices that are generated will receive automatic tax calculation.

For this feature to work properly, your customer's billing details, such as the customer's name, address, and tax ID, need to be synced to Stripe. You may use the customer data synchronization and Tax ID methods offered by Cashier to accomplish this.

{note} Unfortunately, for now, no tax is calculated for single charges or single charge checkouts. In addition, Stripe Tax is currently "invite-only" during its beta period. You can request access to Stripe Tax via the Stripe Tax website.

Logging

Cashier allows you to specify the log channel to be used when logging fatal Stripe errors. You may specify the log channel by defining the `CASHIER_LOGGER` environment variable within your application's `.env` file:

```
CASHIER_LOGGER=stack
```

Exceptions that are generated by API calls to Stripe will be logged through your application's default log channel.

Using Custom Models

You are free to extend the models used internally by Cashier by defining your own model

and extending the corresponding Cashier model:

```
use Laravel\Cashier\Subscription as CashierSubscription;

class Subscription extends CashierSubscription
{
    // ...
}
```

After defining your model, you may instruct Cashier to use your custom model via the `Laravel\Cashier\Cashier` class. Typically, you should inform Cashier about your custom models in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use App\Models\Cashier\Subscription;
use App\Models\Cashier\SubscriptionItem;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Cashier::useSubscriptionModel(Subscription::class);
    Cashier::useSubscriptionItemModel(SubscriptionItem::class);
}
```

Customers

Retrieving Customers

You can retrieve a customer by their Stripe ID using the `Cashier::findBillable` method. This method will return an instance of the billable model:

```
use Laravel\Cashier\Cashier;

$user = Cashier::findBillable($stripeId);
```

Creating Customers

Occasionally, you may wish to create a Stripe customer without beginning a subscription. You may accomplish this using the `createAsStripeCustomer` method:

```
$stripeCustomer = $user->createAsStripeCustomer();
```

Once the customer has been created in Stripe, you may begin a subscription at a later date. You may provide an optional `$options` array to pass in any additional customer creation parameters that are supported by the Stripe API:

```
$stripeCustomer = $user->createAsStripeCustomer($options);
```

You may use the `asStripeCustomer` method if you want to return the Stripe customer object for a billable model:

```
$stripeCustomer = $user->asStripeCustomer();
```

The `createOrGetStripeCustomer` method may be used if you would like to retrieve the Stripe customer object for a given billable model but are not sure whether the billable model

is already a customer within Stripe. This method will create a new customer in Stripe if one does not already exist:

```
$stripeCustomer = $user->createOrGetStripeCustomer();
```

Updating Customers

Occasionally, you may wish to update the Stripe customer directly with additional information. You may accomplish this using the `updateStripeCustomer` method. This method accepts an array of customer update options supported by the Stripe API:

```
$stripeCustomer = $user->updateStripeCustomer($options);
```

Balances

Stripe allows you to credit or debit a customer's "balance". Later, this balance will be credited or debited on new invoices. To check the customer's total balance you may use the `balance` method that is available on your billable model. The `balance` method will return a formatted string representation of the balance in the customer's currency:

```
$balance = $user->balance();
```

To credit a customer's balance, you may provide a negative value to the `applyBalance` method. If you wish, you may also provide a description:

```
$user->applyBalance(-500, 'Premium customer top-up.');
```

Providing a positive value to the `applyBalance` method will debit the customer's balance:

```
$user->applyBalance(300, 'Bad usage penalty.');
```

The `applyBalance` method will create new customer balance transactions for the

customer. You may retrieve these transaction records using the `balanceTransactions` method, which may be useful in order to provide a log of credits and debits for the customer to review:

```
// Retrieve all transactions...
$transactions = $user->balanceTransactions();

foreach ($transactions as $transaction) {
    // Transaction amount...
    $amount = $transaction->amount(); // $2.31

    // Retrieve the related invoice when available...
    $invoice = $transaction->invoice();
}
```

Tax IDs

Cashier offers an easy way to manage a customer's tax IDs. For example, the `taxIds` method may be used to retrieve all of the tax IDs that are assigned to a customer as a collection:

```
$taxIds = $user->taxIds();
```

You can also retrieve a specific tax ID for a customer by its identifier:

```
$taxId = $user->findTaxId('txi_belgium');
```

You may create a new Tax ID by providing a valid type and value to the `createTaxId` method:

```
$taxId = $user->createTaxId('eu_vat', 'BE0123456789');
```

The `createTaxId` method will immediately add the VAT ID to the customer's account. Verification of VAT IDs is also done by Stripe; however, this is an asynchronous process. You can be notified of verification updates by subscribing to the `customer.tax_id.updated` webhook event and inspecting the VAT IDs `verification` parameter. For more information

on handling webhooks, please consult the documentation on defining webhook handlers.

You may delete a tax ID using the `deleteTaxId` method:

```
$user->deleteTaxId('txi_belgium');
```

Syncing Customer Data With Stripe

Typically, when your application's users update their name, email address, or other information that is also stored by Stripe, you should inform Stripe of the updates. By doing so, Stripe's copy of the information will be in sync with your application's.

To automate this, you may define an event listener on your billable model that reacts to the model's `updated` event. Then, within your event listener, you may invoke the `syncStripeCustomerDetails` method on the model:

```
use function Illuminate\Events\queueable;

/**
 * The "booted" method of the model.
 *
 * @return void
 */
protected static function booted()
{
    static::updated(queueable(function ($customer) {
        $customer->syncStripeCustomerDetails();
    }));
}
```

Now, every time your customer model is updated, its information will be synced with Stripe. For convenience, Cashier will automatically sync your customer's information with Stripe on the initial creation of the customer.

You may customize the columns used for syncing customer information to Stripe by overriding a variety of methods provided by Cashier. For example, you may override the `stripeName` method to customize the attribute that should be considered the customer's "name" when Cashier syncs customer information to Stripe:

```
/**
 * Get the customer name that should be synced to Stripe.
 *
 * @return string|null
 */
public function stripeName()
{
    return $this->company_name;
}
```

Similarly, you may override the `stripeEmail`, `stripePhone`, and `stripeAddress` methods. These methods will sync information to their corresponding customer parameters when updating the Stripe customer object. If you wish to take total control over the customer information sync process, you may override the `syncStripeCustomerDetails` method.

Billing Portal

Stripe offers an easy way to set up a billing portal so that your customer can manage their subscription, payment methods, and view their billing history. You can redirect your users to the billing portal by invoking the `redirectToBillingPortal` method on the billable model from a controller or route:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal();
});
```

By default, when the user is finished managing their subscription, they will be able to return to the `home` route of your application via a link within the Stripe billing portal. You may provide a custom URL that the user should return to by passing the URL as an argument to the `redirectToBillingPortal` method:

```
use Illuminate\Http\Request;

Route::get('/billing-portal', function (Request $request) {
    return $request->user()->redirectToBillingPortal(route('billing'));
});
```

If you would like to generate the URL to the billing portal without generating an HTTP redirect response, you may invoke the `billingPortalUrl` method:

```
$url = $request->user()->billingPortalUrl(route('billing'));
```

Payment Methods

Storing Payment Methods

In order to create subscriptions or perform "one off" charges with Stripe, you will need to store a payment method and retrieve its identifier from Stripe. The approach used to accomplish this differs based on whether you plan to use the payment method for subscriptions or single charges, so we will examine both below.

Payment Methods For Subscriptions

When storing a customer's credit card information for future use by a subscription, the Stripe "Setup Intents" API must be used to securely gather the customer's payment method details. A "Setup Intent" indicates to Stripe the intention to charge a customer's payment method. Cashier's `Billable` trait includes the `createSetupIntent` method to easily create a new Setup Intent. You should invoke this method from the route or controller that will render the form which gathers your customer's payment method details:

```
return view('update-payment-method', [
    'intent' => $user->createSetupIntent()
]);
```

After you have created the Setup Intent and passed it to the view, you should attach its secret to the element that will gather the payment method. For example, consider this "update payment method" form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button" data-secret="{{ $intent->client_secret }}">
    Update Payment Method
</button>
```

Next, the Stripe.js library may be used to attach a Stripe Element to the form and securely gather the customer's payment details:

```
<script src="https://js.stripe.com/v3/"></script>

<script>
    const stripe = Stripe('stripe-public-key');

    const elements = stripe.elements();
    const cardElement = elements.create('card');

    cardElement.mount('#card-element');
</script>
```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using Stripe's `confirmCardSetup` method:

```
const cardHolderName = document.getElementById('card-holder-name');
const cardButton = document.getElementById('card-button');
const clientSecret = cardButton.dataset.secret;

cardButton.addEventListener('click', async (e) => {
    const { setupIntent, error } = await stripe.confirmCardSetup(
        clientSecret, {
            payment_method: {
                card: cardElement,
                billing_details: { name: cardHolderName.value }
            }
        }
    );

    if (error) {
        // Display "error.message" to the user...
    } else {
        // The card has been verified successfully...
    }
});
```

After the card has been verified by Stripe, you may pass the resulting `setupIntent.payment_method` identifier to your Laravel application, where it can be attached to the customer. The payment method can either be added as a new payment method or used to update the default payment method. You can also immediately use the payment method identifier to create a new subscription.

{tip} If you would like more information about Setup Intents and gathering customer payment details please review this overview provided by Stripe.

Payment Methods For Single Charges

Of course, when making a single charge against a customer's payment method, we will only need to use a payment method identifier once. Due to Stripe limitations, you may not use the stored default payment method of a customer for single charges. You must allow the customer to enter their payment method details using the Stripe.js library. For example, consider the following form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button">
    Process Payment
</button>
```

After defining such a form, the Stripe.js library may be used to attach a Stripe Element to the form and securely gather the customer's payment details:

```
<script src="https://js.stripe.com/v3/"></script>

<script>
    const stripe = Stripe('stripe-public-key');

    const elements = stripe.elements();
    const cardElement = elements.create('card');

    cardElement.mount('#card-element');
</script>
```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using Stripe's `createPaymentMethod` method:

```
const cardHolderName = document.getElementById('card-holder-name');
const cardButton = document.getElementById('card-button');

cardButton.addEventListener('click', async (e) => {
    const { paymentMethod, error } = await stripe.createPaymentMethod(
        'card', cardElement, {
            billing_details: { name: cardHolderName.value }
        }
    );

    if (error) {
        // Display "error.message" to the user...
    } else {
        // The card has been verified successfully...
    }
});
```

If the card is verified successfully, you may pass the `paymentMethod.id` to your Laravel application and process a single charge.

Retrieving Payment Methods

The `paymentMethods` method on the billable model instance returns a collection of `Laravel\Cashier\PaymentMethod` instances:

```
$paymentMethods = $user->paymentMethods();
```

By default, this method will return payment methods of the `card` type. To retrieve payment methods of a different type, you may pass the `type` as an argument to the method:

```
$paymentMethods = $user->paymentMethods('sepa_debit');
```

To retrieve the customer's default payment method, the `defaultPaymentMethod` method may be used:

```
$paymentMethod = $user->defaultPaymentMethod();
```

You can retrieve a specific payment method that is attached to the billable model using the `findPaymentMethod` method:

```
$paymentMethod = $user->findPaymentMethod($paymentMethodId);
```

Determining If A User Has A Payment Method

To determine if a billable model has a default payment method attached to their account, invoke the `hasDefaultPaymentMethod` method:

```
if ($user->hasDefaultPaymentMethod()) {
    //
}
```

You may use the `hasPaymentMethod` method to determine if a billable model has at least one payment method attached to their account:

```
if ($user->hasPaymentMethod()) {
    //
}
```

This method will determine if the billable model has payment methods of the `card` type. To determine if a payment method of another type exists for the model, you may pass the `type` as an argument to the method:

```
if ($user->hasPaymentMethod('sepa_debit')) {
    //
}
```

Updating The Default Payment Method

The `updateDefaultPaymentMethod` method may be used to update a customer's default payment method information. This method accepts a Stripe payment method identifier and will assign the new payment method as the default billing payment method:

```
$user->updateDefaultPaymentMethod($paymentMethod);
```

To sync your default payment method information with the customer's default payment method information in Stripe, you may use the `updateDefaultPaymentMethodFromStripe` method:

```
$user->updateDefaultPaymentMethodFromStripe();
```

{note} The default payment method on a customer can only be used for invoicing and creating new subscriptions. Due to limitations imposed by Stripe, it may not be used for single charges.

Adding Payment Methods

To add a new payment method, you may call the `addPaymentMethod` method on the billable model, passing the payment method identifier:

```
$user->addPaymentMethod($paymentMethod);
```

{tip} To learn how to retrieve payment method identifiers please review the payment method storage documentation.

Deleting Payment Methods

To delete a payment method, you may call the `delete` method on the `Laravel\Cashier\PaymentMethod` instance you wish to delete:

```
$paymentMethod->delete();
```

The `deletePaymentMethods` method will delete all of the payment method information for the billable model:

```
$user->deletePaymentMethods();
```

By default, this method will delete payment methods of the `card` type. To delete payment methods of a different type you can pass the `type` as an argument to the method:

```
$user->deletePaymentMethods('sepa_debit');
```

{note} If a user has an active subscription, your application should not allow them to delete their default payment method.

Subscriptions

Subscriptions provide a way to set up recurring payments for your customers. Stripe subscriptions managed by Cashier provide support for multiple subscription prices, subscription quantities, trials, and more.

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\Models\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription(
        'default', 'price_monthly'
    )->create($request->paymentMethodId);

    // ...
});
```

The first argument passed to the `newSubscription` method should be the internal name of the subscription. If your application only offers a single subscription, you might call this `default` or `primary`. This subscription name is only for internal application usage and is not meant to be shown to users. In addition, it should not contain spaces and it should never be changed after creating the subscription. The second argument is the specific price the user is subscribing to. This value should correspond to the price's identifier in Stripe.

The `create` method, which accepts a Stripe payment method identifier or Stripe `PaymentMethod` object, will begin the subscription as well as update your database with the billable model's Stripe customer ID and other relevant billing information.

{note} Passing a payment method identifier directly to the `create` subscription method will also automatically add it to the user's stored payment methods.

Collecting Recurring Payments Via Invoice Emails

Instead of collecting a customer's recurring payments automatically, you may instruct Stripe to email an invoice to the customer each time their recurring payment is due. Then, the customer may manually pay the invoice once they receive it. The customer does not need to provide a payment method up front when collecting recurring payments via invoices:

```
$user->newSubscription('default',
'price_monthly')->createAndSendInvoice();
```

The amount of time a customer has to pay their invoice before their subscription is canceled is determined by your subscription and invoice settings within the Stripe dashboard.

Quantities

If you would like to set a specific quantity for the price when creating the subscription, you should invoke the `quantity` method on the subscription builder before creating the subscription:

```
$user->newSubscription('default', 'price_monthly')
     ->quantity(5)
     ->create($paymentMethod);
```

Additional Details

If you would like to specify additional customer or subscription options supported by Stripe, you may do so by passing them as the second and third arguments to the `create` method:

```
$user->newSubscription('default',
'price_monthly')->create($paymentMethod, [
    'email' => $email,
], [
    'metadata' => ['note' => 'Some extra information.'],
]);
```

Coupons

If you would like to apply a coupon when creating the subscription, you may use the

`withCoupon` method:

```
$user->newSubscription('default', 'price_monthly')
     ->withCoupon('code')
     ->create($paymentMethod);
```

Or, if you would like to apply a Stripe promotion code, you may use the `withPromotionCode` method. The given promotion code ID should be the Stripe API ID assigned to the promotion code and not the customer facing promotion code:

```
$user->newSubscription('default', 'price_monthly')
     ->withPromotionCode('promo_code')
     ->create($paymentMethod);
```

Adding Subscriptions

If you would like to add a subscription to a customer who already has a default payment method you may invoke the `add` method on the subscription builder:

```
use App\Models\User;

$user = User::find(1);

$user->newSubscription('default', 'price_monthly')->add();
```

Creating Subscriptions From The Stripe Dashboard

You may also create subscriptions from the Stripe dashboard itself. When doing so, Cashier will sync newly added subscriptions and assign them a name of `default`. To customize the subscription name that is assigned to dashboard created subscriptions, extend the `WebhookController` and overwrite the `newSubscriptionName` method.

In addition, you may only create one type of subscription via the Stripe dashboard. If your application offers multiple subscriptions that use different names, only one type of subscription may be added through the Stripe dashboard.

Finally, you should always make sure to only add one active subscription per type of subscription offered by your application. If customer has two `default` subscriptions, only the most recently added subscription will be used by Cashier even though both would be

synced with your application's database.

Checking Subscription Status

Once a customer is subscribed to your application, you may easily check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the customer has an active subscription, even if the subscription is currently within its trial period. The `subscribed` method accepts the name of the subscription as its first argument:

```
if ($user->subscribed('default')) {
    //
}
```

The `subscribed` method also makes a great candidate for a route middleware, allowing you to filter access to routes and controllers based on the user's subscription status:

```
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureUserIsSubscribed
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->user() && !
$request->user()->subscribed('default')) {
            // This user is not a paying customer...
            return redirect('billing');
        }

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

If you would like to determine if a user is still within their trial period, you may use the `onTrial` method. This method can be useful for determining if you should display a warning to the user that they are still on their trial period:

```
if ($user->subscription('default')->onTrial()) {
    //
}
```

The `subscribedToProduct` method may be used to determine if the user is subscribed to a given product based on a given Stripe product's identifier. In Stripe, products are collections of prices. In this example, we will determine if the user's `default` subscription is actively subscribed to the application's "premium" product. The given Stripe product identifier should correspond to one of your product's identifiers in the Stripe dashboard:

```
if ($user->subscribedToProduct('prod_premium', 'default')) {
    //
}
```

By passing an array to the `subscribedToProduct` method, you may determine if the user's `default` subscription is actively subscribed to the application's "basic" or "premium" product:

```
if ($user->subscribedToProduct(['prod_basic', 'prod_premium'],
'default')) {
    //
}
```

The `subscribedToPrice` method may be used to determine if a customer's subscription corresponds to a given price ID:

```
if ($user->subscribedToPrice('price_basic_monthly', 'default')) {
    //
}
```

The `recurring` method may be used to determine if the user is currently subscribed and is no longer within their trial period:

```
if ($user->subscription('default')->recurring()) {
    //
}
```

{note} If a user has two subscriptions with the same name, the most recent subscription will always be returned by the `subscription` method. For example, a user might have two subscription records named `default`; however, one of the subscriptions may be an old, expired subscription, while the other is the current, active subscription. The most recent subscription will always be returned while older subscriptions are kept in the database for historical review.

Canceled Subscription Status

To determine if the user was once an active subscriber but has canceled their subscription,

you may use the `canceled` method:

```
if ($user->subscription('default')->canceled()) {
    //
}
```

You may also determine if a user has canceled their subscription but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. Note that the `subscribed` method still returns `true` during this time:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

To determine if the user has canceled their subscription and is no longer within their "grace period", you may use the `ended` method:

```
if ($user->subscription('default')->ended()) {
    //
}
```

Incomplete and Past Due Status

If a subscription requires a secondary payment action after creation the subscription will be marked as `incomplete`. Subscription statuses are stored in the `stripe_status` column of Cashier's `subscriptions` database table.

Similarly, if a secondary payment action is required when swapping prices the subscription will be marked as `past_due`. When your subscription is in either of these states it will not be active until the customer has confirmed their payment. Determining if a subscription has an incomplete payment may be accomplished using the `hasIncompletePayment` method on the billable model or a subscription instance:

```
if ($user->hasIncompletePayment('default')) {
    //
}

if ($user->subscription('default')->hasIncompletePayment()) {
    //
}
```

When a subscription has an incomplete payment, you should direct the user to Cashier's payment confirmation page, passing the `latestPayment` identifier. You may use the `latestPayment` method available on subscription instance to retrieve this identifier:

```
<a href="{{ route('cashier.payment', $subscription->latestPayment()->id) }}">
    Please confirm your payment.
</a>
```

If you would like the subscription to still be considered active when it's in a `past_due` state, you may use the `keepPastDueSubscriptionsActive` method provided by Cashier. Typically, this method should be called in the `register` method of your `App\Providers\AppServiceProvider`:

```
use Laravel\Cashier\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::keepPastDueSubscriptionsActive();
}
```

{note} When a subscription is in an `incomplete` state it cannot be changed until the payment is confirmed. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in an `incomplete` state.

Subscription Scopes

Most subscription states are also available as query scopes so that you may easily query your database for subscriptions that are in a given state:

```
// Get all active subscriptions...
$subscriptions = Subscription::query()->active()->get();

// Get all of the canceled subscriptions for a user...
$subscriptions = $user->subscriptions()->canceled()->get();
```

A complete list of available scopes is available below:

```
Subscription::query()->active();
Subscription::query()->canceled();
Subscription::query()->ended();
Subscription::query()->incomplete();
Subscription::query()->notCanceled();
Subscription::query()->notOnGracePeriod();
Subscription::query()->notOnTrial();
Subscription::query()->onGracePeriod();
Subscription::query()->onTrial();
Subscription::query()->pastDue();
Subscription::query()->recurring();
```

Changing Prices

After a customer is subscribed to your application, they may occasionally want to change to a new subscription price. To swap a customer to a new price, pass the Stripe price's identifier to the swap method. When swapping prices, it is assumed that the user would like to re-activate their subscription if it was previously canceled. The given price identifier should correspond to a Stripe price identifier available in the Stripe dashboard:

```
use App\Models\User;

$user = App\Models\User::find(1);

$user->subscription('default')->swap('price_yearly');
```

If the customer is on trial, the trial period will be maintained. Additionally, if a "quantity" exists for the subscription, that quantity will also be maintained.

If you would like to swap prices and cancel any trial period the customer is currently on, you may invoke the `skipTrial` method:

```
$user->subscription('default')
        ->skipTrial()
        ->swap('price_yearly');
```

If you would like to swap prices and immediately invoice the customer instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = User::find(1);

$user->subscription('default')->swapAndInvoice('price_yearly');
```

Prorations

By default, Stripe prorates charges when swapping between prices. The `noProrate` method may be used to update the subscription's price without prorating the charges:

```
$user->subscription('default')->noProrate()->swap('price_yearly');
```

For more information on subscription proration, consult the Stripe documentation.

{note} Executing the `noProrate` method before the `swapAndInvoice` method will have no effect on proration. An invoice will always be issued.

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, a project management application might charge $10 per month per project. You may use the `incrementQuantity` and `decrementQuantity` methods to easily increment or decrement your subscription quantity:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('default')->incrementQuantity(5);

$user->subscription('default')->decrementQuantity();

// Subtract five from the subscription's current quantity...
$user->subscription('default')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('default')->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without prorating the charges:

```
$user->subscription('default')->noProrate()->updateQuantity(10);
```

For more information on subscription quantities, consult the Stripe documentation.

Multiprice Subscription Quantities

If your subscription is a multiprice subscription, you should pass the name of the price whose quantity you wish to increment or decrement as the second argument to the increment / decrement methods:

```
$user->subscription('default')->incrementQuantity(1, 'price_chat');
```

Multiprice Subscriptions

Multiprice subscriptions allow you to assign multiple billing prices to a single subscription. For example, imagine you are building a customer service "helpdesk" application that has a base subscription price of $10 per month but offers a live chat add-on price for an additional $15 per month. Multiprice subscription information is stored in Cashier's `subscription_items` database table.

You may specify multiple prices for a given subscription by passing an array of prices as the second argument to the `newSubscription` method:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default', [
        'price_monthly',
        'price_chat',
    ])->create($request->paymentMethodId);

    // ...
});
```

In the example above, the customer will have two prices attached to their `default` subscription. Both prices will be charged on their respective billing intervals. If necessary, you may use the `quantity` method to indicate a specific quantity for each price:

```
$user = User::find(1);

$user->newSubscription('default', ['price_monthly', 'price_chat'])
    ->quantity(5, 'price_chat')
    ->create($paymentMethod);
```

If you would like to add another price to an existing subscription, you may invoke the subscription's `addPrice` method:

```
$user = User::find(1);

$user->subscription('default')->addPrice('price_chat');
```

The example above will add the new price and the customer will be billed for it on their next billing cycle. If you would like to bill the customer immediately you may use the `addPriceAndInvoice` method:

```
$user->subscription('default')->addPriceAndInvoice('price_chat');
```

If you would like to add a price with a specific quantity, you can pass the quantity as the second argument of the `addPrice` or `addPriceAndInvoice` methods:

```
$user = User::find(1);

$user->subscription('default')->addPrice('price_chat', 5);
```

You may remove prices from subscriptions using the `removePrice` method:

```
$user->subscription('default')->removePrice('price_chat');
```

{note} You may not remove the last price on a subscription. Instead, you should simply cancel the subscription.

Swapping Prices

You may also change the prices attached to a multiprice subscription. For example, imagine a customer has a `price_basic` subscription with a `price_chat` add-on price and you want to upgrade the customer from the `price_basic` to the `price_pro` price:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->swap(['price_pro', 'price_chat']);
```

When executing the example above, the underlying subscription item with the `price_basic` is deleted and the one with the `price_chat` is preserved. Additionally, a new subscription item for the `price_pro` is created.

You can also specify subscription item options by passing an array of key / value pairs to the `swap` method. For example, you may need to specify the subscription price quantities:

```
$user = User::find(1);

$user->subscription('default')->swap([
    'price_pro' => ['quantity' => 5],
    'price_chat'
]);
```

If you want to swap a single price on a subscription, you may do so using the `swap` method on the subscription item itself. This approach is particularly useful if you would like to preserve all of the existing metadata on the subscription's other prices:

```
$user = User::find(1);

$user->subscription('default')
        ->findItemOrFail('price_basic')
        ->swap('price_pro');
```

Proration

By default, Stripe will prorate charges when adding or removing prices from a multiprice subscription. If you would like to make a price adjustment without proration, you should chain the `noProrate` method onto your price operation:

```
$user->subscription('default')->noProrate()->removePrice('price_chat');
```

Quantities

If you would like to update quantities on individual subscription prices, you may do so using the existing quantity methods by passing the name of the price as an additional argument to the method:

```
$user = User::find(1);

$user->subscription('default')->incrementQuantity(5, 'price_chat');

$user->subscription('default')->decrementQuantity(3, 'price_chat');

$user->subscription('default')->updateQuantity(10, 'price_chat');
```

{note} When a subscription has multiple prices the `stripe_price` and `quantity` attributes on the `Subscription` model will be `null`. To access the individual price attributes, you should use the `items` relationship available on the `Subscription` model.

Subscription Items

When a subscription has multiple prices, it will have multiple subscription "items" stored in your database's `subscription_items` table. You may access these via the `items` relationship on the subscription:

```
use App\Models\User;

$user = User::find(1);

$subscriptionItem = $user->subscription('default')->items->first();

// Retrieve the Stripe price and quantity for a specific item...
$stripePrice = $subscriptionItem->stripe_price;
$quantity = $subscriptionItem->quantity;
```

You can also retrieve a specific price using the `findItemOrFail` method:

```
$user = User::find(1);

$subscriptionItem =
$user->subscription('default')->findItemOrFail('price_chat');
```

Metered Billing

Metered billing allows you to charge customers based on their product usage during a billing cycle. For example, you may charge customers based on the number of text messages or emails they send per month.

To start using metered billing, you will first need to create a new product in your Stripe dashboard with a metered price. Then, use the `meteredPrice` to add the metered price ID to a customer subscription:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default')
        ->meteredPrice('price_metered')
        ->create($request->paymentMethodId);

    // ...
});
```

You may also start a metered subscription via Stripe Checkout:

```
$checkout = Auth::user()
        ->newSubscription('default', [])
        ->meteredPrice('price_metered')
        ->checkout();

return view('your-checkout-view', [
    'checkout' => $checkout,
]);
```

Reporting Usage

As your customer uses your application, you will report their usage to Stripe so that they can be billed accurately. To increment the usage of a metered subscription, you may use the `reportUsage` method:

```
$user = User::find(1);

$user->subscription('default')->reportUsage();
```

By default, a "usage quantity" of 1 is added to the billing period. Alternatively, you may pass a specific amount of "usage" to add to the customer's usage for the billing period:

```
$user = User::find(1);

$user->subscription('default')->reportUsage(15);
```

If your application offers multiple prices on a single subscription, you will need to use the `reportUsageFor` method to specify the metered price you want to report usage for:

```
$user = User::find(1);

$user->subscription('default')->reportUsageFor('price_metered', 15);
```

Sometimes, you may need to update usage which you have previously reported. To accomplish this, you may pass a timestamp or a `DateTimeInterface` instance as the second parameter to `reportUsage`. When doing so, Stripe will update the usage that was reported at that given time. You can continue to update previous usage records as the given date and time is still within the current billing period:

```
$user = User::find(1);

$user->subscription('default')->reportUsage(5, $timestamp);
```

Retrieving Usage Records

To retrieve a customer's past usage, you may use a subscription instance's `usageRecords` method:

```
$user = User::find(1);

$usageRecords = $user->subscription('default')->usageRecords();
```

If your application offers multiple prices on a single subscription, you may use the `usageRecordsFor` method to specify the metered price that you wish to retrieve usage records for:

```
$user = User::find(1);

$usageRecords =
$user->subscription('default')->usageRecordsFor('price_metered');
```

The `usageRecords` and `usageRecordsFor` methods return a Collection instance containing an associative array of usage records. You may iterate over this array to display a customer's total usage:

```
@foreach ($usageRecords as $usageRecord)
    - Period Starting: {{ $usageRecord['period']['start'] }}
    - Period Ending: {{ $usageRecord['period']['end'] }}
    - Total Usage: {{ $usageRecord['total_usage'] }}
@endforeach
```

For a full reference of all usage data returned and how to use Stripe's cursor based pagination, please consult the official Stripe API documentation.

Subscription Taxes

{note} Instead of calculating Tax Rates manually, you can automatically calculate taxes using Stripe Tax

To specify the tax rates a user pays on a subscription, you should implement the `taxRates` method on your billable model and return an array containing the Stripe tax rate IDs. You can define these tax rates in your Stripe dashboard:

```
/**
 * The tax rates that should apply to the customer's subscriptions.
 *
 * @return array
 */
public function taxRates()
{
    return ['txr_id'];
}
```

The `taxRates` method enables you to apply a tax rate on a customer-by-customer basis, which may be helpful for a user base that spans multiple countries and tax rates.

If you're offering multiprice subscriptions, you may define different tax rates for each price by implementing a `priceTaxRates` method on your billable model:

```
/**
 * The tax rates that should apply to the customer's subscriptions.
 *
 * @return array
 */
public function priceTaxRates()
{
    return [
        'price_monthly' => ['txr_id'],
    ];
}
```

{note} The `taxRates` method only applies to subscription charges. If you use Cashier to make "one off" charges, you will need to manually specify the tax rate at that time.

Syncing Tax Rates

When changing the hard-coded tax rate IDs returned by the `taxRates` method, the tax settings on any existing subscriptions for the user will remain the same. If you wish to update the tax value for existing subscriptions with the new `taxRates` values, you should call the `syncTaxRates` method on the user's subscription instance:

```
$user->subscription('default')->syncTaxRates();
```

This will also sync any multiprice subscription item tax rates. If your application is offering multiprice subscriptions, you should ensure that your billable model implements the `priceTaxRates` method discussed above.

Tax Exemption

Cashier also offers the `isNotTaxExempt`, `isTaxExempt`, and `reverseChargeApplies` methods to determine if the customer is tax exempt. These methods will call the Stripe API to determine a customer's tax exemption status:

```
use App\Models\User;

$user = User::find(1);

$user->isTaxExempt();
$user->isNotTaxExempt();
$user->reverseChargeApplies();
```

{note} These methods are also available on any `Laravel\Cashier\Invoice` object. However, when invoked on an `Invoice` object, the methods will determine the exemption status at the time the invoice was created.

Subscription Anchor Date

By default, the billing cycle anchor is the date the subscription was created or, if a trial period is used, the date that the trial ends. If you would like to modify the billing anchor date, you may use the `anchorBillingCycleOn` method:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $anchor = Carbon::parse('first day of next month');

    $request->user()->newSubscription('default', 'price_monthly')
                ->anchorBillingCycleOn($anchor->startOfDay())
                ->create($request->paymentMethodId);

    // ...
});
```

For more information on managing subscription billing cycles, consult the Stripe billing cycle documentation

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription('default')->cancel();
```

When a subscription is canceled, Cashier will automatically set the `ends_at` column in your `subscriptions` database table. This column is used to know when the `subscribed` method should begin returning `false`.

For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has canceled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

If you wish to cancel a subscription immediately, call the `cancelNow` method on the user's

subscription:

```
$user->subscription('default')->cancelNow();
```

If you wish to cancel a subscription immediately and invoice any remaining un-invoiced metered usage or new / pending proration invoice items, call the `cancelNowAndInvoice` method on the user's subscription:

```
$user->subscription('default')->cancelNowAndInvoice();
```

You may also choose to cancel the subscription at a specific moment in time:

```
$user->subscription('default')->cancelAt(
    now()->addDays(10)
);
```

Resuming Subscriptions

If a customer has canceled their subscription and you wish to resume it, you may invoke the `resume` method on the subscription. The customer must still be within their "grace period" in order to resume a subscription:

```
$user->subscription('default')->resume();
```

If the customer cancels a subscription and then resumes that subscription before the subscription has fully expired the customer will not be billed immediately. Instead, their subscription will be re-activated and they will be billed on the original billing cycle.

Subscription Trials

With Payment Method Up Front

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should use the `trialDays` method when creating your subscriptions:

```
use Illuminate\Http\Request;

Route::post('/user/subscribe', function (Request $request) {
    $request->user()->newSubscription('default', 'price_monthly')
                ->trialDays(10)
                ->create($request->paymentMethodId);

    // ...
});
```

This method will set the trial period ending date on the subscription record within the database and instruct Stripe to not begin billing the customer until after this date. When using the `trialDays` method, Cashier will overwrite any default trial period configured for the price in Stripe.

{note} If the customer's subscription is not canceled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

The `trialUntil` method allows you to provide a `DateTime` instance that specifies when the trial period should end:

```
use Carbon\Carbon;

$user->newSubscription('default', 'price_monthly')
            ->trialUntil(Carbon::now()->addDays(10))
            ->create($paymentMethod);
```

You may determine if a user is within their trial period using either the `onTrial` method of

the user instance or the `onTrial` method of the subscription instance. The two examples below are equivalent:

```
if ($user->onTrial('default')) {
    //
}

if ($user->subscription('default')->onTrial()) {
    //
}
```

You may use the `endTrial` method to immediately end a subscription trial:

```
$user->subscription('default')->endTrial();
```

Defining Trial Days In Stripe / Cashier

You may choose to define how many trial days your price's receive in the Stripe dashboard or always pass them explicitly using Cashier. If you choose to define your price's trial days in Stripe you should be aware that new subscriptions, including new subscriptions for a customer that had a subscription in the past, will always receive a trial period unless you explicitly call the `skipTrial()` method.

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the user record to your desired trial ending date. This is typically done during user registration:

```
use App\Models\User;

$user = User::create([
    // ...
    'trial_ends_at' => now()->addDays(10),
]);
```

{note} Be sure to add a date cast for the `trial_ends_at` attribute within your billable model's class definition.

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the billable model instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
$user = User::find(1);

$user->newSubscription('default',
'price_monthly')->create($paymentMethod);
```

To retrieve the user's trial ending date, you may use the `trialEndsAt` method. This method will return a Carbon date instance if a user is on a trial or `null` if they aren't. You may also pass an optional subscription name parameter if you would like to get the trial ending date for a specific subscription other than the default one:

```
if ($user->onTrial()) {
    $trialEndsAt = $user->trialEndsAt('main');
}
```

You may also use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not yet created an actual subscription:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

Extending Trials

The `extendTrial` method allows you to extend the trial period of a subscription after the subscription has been created. If the trial has already expired and the customer is already being billed for the subscription, you can still offer them an extended trial. The time spent within the trial period will be deducted from the customer's next invoice:

```
use App\Models\User;

$subscription = User::find(1)->subscription('default');

// End the trial 7 days from now...
$subscription->extendTrial(
    now()->addDays(7)
);

// Add an additional 5 days to the trial...
$subscription->extendTrial(
    $subscription->trial_ends_at->addDays(5)
);
```

Handling Stripe Webhooks

{tip} You may use the Stripe CLI to help test webhooks during local development.

Stripe can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is automatically registered by the Cashier service provider. This controller will handle all incoming webhook requests.

By default, the Cashier webhook controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Stripe settings), customer updates, customer deletions, subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any Stripe webhook event you like.

To ensure your application can handle Stripe webhooks, be sure to configure the webhook URL in the Stripe control panel. By default, Cashier's webhook controller responds to the `/stripe/webhook` URL path. The full list of all webhooks you should enable in the Stripe control panel are:

- `customer.subscription.created`
- `customer.subscription.updated`
- `customer.subscription.deleted`
- `customer.updated`
- `customer.deleted`
- `invoice.payment_action_required`

For convenience, Cashier includes a `cashier:webhook` Artisan command. This command will create a webhook in Stripe that listens to all of the events required by Cashier:

```
php artisan cashier:webhook
```

By default, the created webhook will point to the URL defined by the `APP_URL` environment variable and the `cashier.webhook` route that is included with Cashier. You may provide the `--url` option when invoking the command if you would like to use a different URL:

```
php artisan cashier:webhook --url "https://example.com/stripe/webhook"
```

The webhook that is created will use the Stripe API version that your version of Cashier is compatible with. If you would like to use a different Stripe version, you may provide the `--api-version` option:

```
php artisan cashier:webhook --api-version="2019-12-03"
```

After creation, the webhook will be immediately active. If you wish to create the webhook but have it disabled until you're ready, you may provide the `--disabled` option when invoking the command:

```
php artisan cashier:webhook --disabled
```

{note} Make sure you protect incoming Stripe webhook requests with Cashier's included webhook signature verification middleware.

Webhooks & CSRF Protection

Since Stripe webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your application's `App\Http\Middleware\VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'stripe/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellations for failed charges and other common Stripe webhook events. However, if you have additional webhook events you would like to handle, you may do so by listening to the following events that are dispatched by Cashier:

- `Laravel\Cashier\Events\WebhookReceived`
- `Laravel\Cashier\Events\WebhookHandled`

Both events contain the full payload of the Stripe webhook. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you may register a listener that will handle the event:

```php
<?php

namespace App\Listeners;

use Laravel\Cashier\Events\WebhookReceived;

class StripeEventListener
{
    /**
     * Handle received Stripe webhooks.
     *
     * @param  \Laravel\Cashier\Events\WebhookReceived  $event
     * @return void
     */
    public function handle(WebhookReceived $event)
    {
        if ($event->payload['type'] === 'invoice.payment_succeeded') {
            // Handle the incoming event...
        }
    }
}
```

Once your listener has been defined, you may register it within your application's `EventServiceProvider`:

```php
<?php

namespace App\Providers;

use App\Listeners\StripeEventListener;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;
use Laravel\Cashier\Events\WebhookReceived;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        WebhookReceived::class => [
            StripeEventListener::class,
        ],
    ];
}
```

Verifying Webhook Signatures

To secure your webhooks, you may use Stripe's webhook signatures. For convenience, Cashier automatically includes a middleware which validates that the incoming Stripe webhook request is valid.

To enable webhook verification, ensure that the `STRIPE_WEBHOOK_SECRET` environment variable is set in your application's `.env` file. The webhook `secret` may be retrieved from your Stripe account dashboard.

Single Charges

Simple Charge

{note} The `charge` method accepts the amount you would like to charge in the lowest denominator of the currency used by your application. For example, when using United States Dollars, amounts should be specified in pennies.

If you would like to make a one-time charge against a customer, you may use the `charge` method on a billable model instance. You will need to provide a payment method identifier as the second argument to the `charge` method:

```
use Illuminate\Http\Request;

Route::post('/purchase', function (Request $request) {
    $stripeCharge = $request->user()->charge(
        100, $request->paymentMethodId
    );

    // ...
});
```

The `charge` method accepts an array as its third argument, allowing you to pass any options you wish to the underlying Stripe charge creation. More information regarding the options available to you when creating charges may be found in the Stripe documentation:

```
$user->charge(100, $paymentMethod, [
    'custom_option' => $value,
]);
```

You may also use the `charge` method without an underlying customer or user. To accomplish this, invoke the `charge` method on a new instance of your application's billable model:

```
use App\Models\User;

$stripeCharge = (new User)->charge(100, $paymentMethod);
```

The `charge` method will throw an exception if the charge fails. If the charge is successful, an instance of `Laravel\Cashier\Payment` will be returned from the method:

```
try {
    $payment = $user->charge(100, $paymentMethod);
} catch (Exception $e) {
    //
}
```

Charge With Invoice

Sometimes you may need to make a one-time charge and offer a PDF receipt to your customer. The `invoicePrice` method lets you do just that. For example, let's invoice a customer for five new shirts:

```
$user->invoicePrice('price_tshirt', 5);
```

The invoice will be immediately charged against the user's default payment method. The `invoicePrice` method also accepts an array as its third argument. This array contains the billing options for the invoice item. The fourth argument accepted by the method is also an array which should contain the billing options for the invoice itself:

```
$user->invoicePrice('price_tshirt', 5, [
    'discounts' => [
        ['coupon' => 'SUMMER21SALE']
    ],
], [
    'default_tax_rates' => ['txr_id'],
]);
```

Alternatively, you may use the `invoiceFor` method to make a "one-off" charge against the customer's default payment method:

```
$user->invoiceFor('One Time Fee', 500);
```

Although the `invoiceFor` method is available for you to use, it is recommendeded that you use the `invoicePrice` method with pre-defined prices. By doing so, you will have access to better analytics and data within your Stripe dashboard regarding your sales on a per-product basis.

{note} The `invoicePrice` and `invoiceFor` methods will create a Stripe invoice which will retry failed billing attempts. If you do not want invoices to retry failed charges, you will need to close them using the Stripe API after the first failed charge.

Refunding Charges

If you need to refund a Stripe charge, you may use the `refund` method. This method accepts the Stripe payment intent ID as its first argument:

```
$payment = $user->charge(100, $paymentMethodId);

$user->refund($payment->id);
```

Invoices

Retrieving Invoices

You may easily retrieve an array of a billable model's invoices using the `invoices` method. The `invoices` method returns a collection of `Laravel\Cashier\Invoice` instances:

```
$invoices = $user->invoices();
```

If you would like to include pending invoices in the results, you may use the `invoicesIncludingPending` method:

```
$invoices = $user->invoicesIncludingPending();
```

You may use the `findInvoice` method to retrieve a specific invoice by its ID:

```
$invoice = $user->findInvoice($invoiceId);
```

Displaying Invoice Information

When listing the invoices for the customer, you may use the invoice's methods to display the relevant invoice information. For example, you may wish to list every invoice in a table, allowing the user to easily download any of them:

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id
}}">Download</a></td>
        </tr>
    @endforeach
</table>
```

Upcoming Invoices

To retrieve the upcoming invoice for a customer, you may use the `upcomingInvoice` method:

```
$invoice = $user->upcomingInvoice();
```

Similary, if the customer has multiple subscriptions, you can also retrieve the upcoming invoice for a specific subscription:

```
$invoice = $user->subscription('default')->upcomingInvoice();
```

Previewing Subscription Invoice

Using the `previewInvoice` method, you can preview an invoice before making price changes. This will allow you to determine what your customer's invoice will look like when a given price change is made:

```
$invoice =
$user->subscription('default')->previewInvoice('price_yearly');
```

You may pass an array of prices to the `previewInvoice` method in order to preview invoices with multiple new prices:

```
$invoice =
$user->subscription('default')->previewInvoice(['price_yearly',
'price_metered']);
```

Generating Invoice PDFs

From within a route or controller, you may use the `downloadInvoice` method to generate a PDF download of a given invoice. This method will automatically generate the proper HTTP response needed to download the invoice:

```
use Illuminate\Http\Request;

Route::get('/user/invoice/{invoice}', function (Request $request,
$invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor' => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

By default, all data on the invoice is derived from the customer and invoice data stored in Stripe. However, you can customize some of this data by providing an array as the second argument to the `downloadInvoice` method. This array allows you to customize information such as your company and product details:

```
return $request->user()->downloadInvoice($invoiceId, [
    'vendor' => 'Your Company',
    'product' => 'Your Product',
    'street' => 'Main Str. 1',
    'location' => '2000 Antwerp, Belgium',
    'phone' => '+32 499 00 00 00',
    'email' => 'info@example.com',
    'url' => 'https://example.com',
    'vendorVat' => 'BE123456789',
], 'my-invoice');
```

The `downloadInvoice` method also allows for a custom filename via its third argument. This filename will automatically be suffixed with `.pdf`:

```
return $request->user()->downloadInvoice($invoiceId, [], 'my-invoice');
```

Checkout

Cashier Stripe also provides support for Stripe Checkout. Stripe Checkout takes the pain out of implementing custom pages to accept payments by providing a pre-built, hosted payment page.

The following documentation contains information on how to get started using Stripe Checkout with Cashier. To learn more about Stripe Checkout, you should also consider reviewing Stripe's own documentation on Checkout.

Product Checkouts

You may perform a checkout for an existing product that has been created within your Stripe dashboard using the `checkout` method on a billable model. The `checkout` method will initiate a new Stripe Checkout session. By default, you're required to pass a Stripe Price ID:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout('price_tshirt');
});
```

If needed, you may also specify a product quantity:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 15]);
});
```

When a customer visits this route they will be redirected to Stripe's Checkout page. By default, when a user successfully completes or cancels a purchase they will be redirected to your `home` route location, but you may specify custom callback URLs using the `success_url` and `cancel_url` options:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 1], [
        'success_url' => route('your-success-route'),
        'cancel_url' => route('your-cancel-route'),
    ]);
});
```

When defining your `success_url` checkout option, you may instruct Stripe to add the checkout session ID as a query string parameter when invoking your URL. To do so, add the literal string `{CHECKOUT_SESSION_ID}` to your `success_url` query string. Stripe will replace this placeholder with the actual checkout session ID:

```
use Illuminate\Http\Request;
use Stripe\Checkout\Session;
use Stripe\Customer;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()->checkout(['price_tshirt' => 1], [
        'success_url' => route('checkout-success') .
'?session_id={CHECKOUT_SESSION_ID}',
        'cancel_url' => route('checkout-cancel'),
    ]);
});

Route::get('/checkout-success', function (Request $request) {
    $checkoutSession =
$request->user()->stripe()->checkout->sessions->retrieve($request->get('
session_id'));

    return view('checkout.success', ['checkoutSession' =>
$checkoutSession]);
})->name('checkout-success');
```

Promotion Codes

By default, Stripe Checkout does not allow user redeemable promotion codes. Luckily, there's an easy way to enable these for your Checkout page. To do so, you may invoke the `allowPromotionCodes` method:

```
use Illuminate\Http\Request;

Route::get('/product-checkout', function (Request $request) {
    return $request->user()
        ->allowPromotionCodes()
        ->checkout('price_tshirt');
});
```

Single Charge Checkouts

You can also perform a simple charge for an ad-hoc product that has not been created in your Stripe dashboard. To do so you may use the `checkoutCharge` method on a billable model and pass it a chargeable amount, a product name, and an optional quantity. When a customer visits this route they will be redirected to Stripe's Checkout page:

```
use Illuminate\Http\Request;

Route::get('/charge-checkout', function (Request $request) {
    return $request->user()->checkoutCharge(1200, 'T-Shirt', 5);
});
```

{note} When using the `checkoutCharge` method, Stripe will always create a new product and price in your Stripe dashboard. Therefore, we recommend that you create the products up front in your Stripe dashboard and use the `checkout` method instead.

Subscription Checkouts

{note} Using Stripe Checkout for subscriptions requires you to enable the `customer.subscription.created` webhook in your Stripe dashboard. This webhook will create the subscription record in your database and store all of the relevant subscription items.

You may also use Stripe Checkout to initiate subscriptions. After defining your subscription with Cashier's subscription builder methods, you may call the `checkout` method. When a customer visits this route they will be redirected to Stripe's Checkout page:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->checkout();
});
```

Just as with product checkouts, you may customize the success and cancellation URLs:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->checkout([
            'success_url' => route('your-success-route'),
            'cancel_url' => route('your-cancel-route'),
        ]);
});
```

Of course, you can also enable promotion codes for subscription checkouts:

```
use Illuminate\Http\Request;

Route::get('/subscription-checkout', function (Request $request) {
    return $request->user()
        ->newSubscription('default', 'price_monthly')
        ->allowPromotionCodes()
        ->checkout();
});
```

{note} Unfortunately Stripe Checkout does not support all subscription billing options when starting subscriptions. Using the `anchorBillingCycleOn` method on the subscription builder, setting proration behavior, or setting payment behavior will not have any effect during Stripe Checkout sessions. Please consult the Stripe Checkout Session API documentation to review which parameters are available.

Stripe Checkout & Trial Periods

Of course, you can define a trial period when building a subscription that will be completed using Stripe Checkout:

```
$checkout = Auth::user()->newSubscription('default', 'price_monthly')
    ->trialDays(3)
    ->checkout();
```

However, the trial period must be at least 48 hours, which is the minimum amount of trial time supported by Stripe Checkout.

Subscriptions & Webhooks

Remember, Stripe and Cashier update subscription statuses via webhooks, so there's a possibility a subscription might not yet be active when the customer returns to the application after entering their payment information. To handle this scenario, you may wish to display a message informing the user that their payment or subscription is pending.

Collecting Tax IDs

Checkout also supports collecting a customer's Tax ID. To enable this on a checkout session, invoke the `collectTaxIds` method when creating the session:

```
$checkout = $user->collectTaxIds()->checkout('price_tshirt');
```

When this method is invoked, a new checkbox will be available to the customer that allows them to indicate if they're purchasing as a company. If so, they will have the opportunity to provide their Tax ID number.

{note} If you have already configured automatic tax collection in your application's service provider then this feature will be enabled automatically and there is no need to invoke the `collectTaxIds` method.

Handling Failed Payments

Sometimes, payments for subscriptions or single charges can fail. When this happens, Cashier will throw an `Laravel\Cashier\Exceptions\IncompletePayment` exception that informs you that this happened. After catching this exception, you have two options on how to proceed.

First, you could redirect your customer to the dedicated payment confirmation page which is included with Cashier. This page already has an associated named route that is registered via Cashier's service provider. So, you may catch the `IncompletePayment` exception and redirect the user to the payment confirmation page:

```
use Laravel\Cashier\Exceptions\IncompletePayment;

try {
    $subscription = $user->newSubscription('default', 'price_monthly')
                         ->create($paymentMethod);
} catch (IncompletePayment $exception) {
    return redirect()->route(
        'cashier.payment',
        [$exception->payment->id, 'redirect' => route('home')]
    );
}
```

On the payment confirmation page, the customer will be prompted to enter their credit card information again and perform any additional actions required by Stripe, such as "3D Secure" confirmation. After confirming their payment, the user will be redirected to the URL provided by the `redirect` parameter specified above. Upon redirection, `message` (string) and `success` (integer) query string variables will be added to the URL. The payment page currently supports the following payment method types:

- Credit Cards - Alipay - Bancontact - BECS Direct Debit - EPS - Giropay - iDEAL - SEPA Direct Debit

Alternatively, you could allow Stripe to handle the payment confirmation for you. In this case, instead of redirecting to the payment confirmation page, you may setup Stripe's automatic billing emails in your Stripe dashboard. However, if an `IncompletePayment` exception is caught, you should still inform the user they will receive an email with further payment confirmation instructions.

Payment exceptions may be thrown for the following methods: `charge`, `invoiceFor`, and `invoice` on models using the `Billable` trait. When interacting with subscriptions, the

`create` method on the `SubscriptionBuilder`, and the `incrementAndInvoice` and `swapAndInvoice` methods on the `Subscription` and `SubscriptionItem` models may throw incomplete payment exceptions.

Determining if an existing subscription has an incomplete payment may be accomplished using the `hasIncompletePayment` method on the billable model or a subscription instance:

```
if ($user->hasIncompletePayment('default')) {
    //
}

if ($user->subscription('default')->hasIncompletePayment()) {
    //
}
```

You can derive the specific status of an incomplete payment by inspecting the `payment` property on the exception instance:

```
use Laravel\Cashier\Exceptions\IncompletePayment;

try {
    $user->charge(1000, 'pm_card_threeDSecure2Required');
} catch (IncompletePayment $exception) {
    // Get the payment intent status...
    $exception->payment->status;

    // Check specific conditions...
    if ($exception->payment->requiresPaymentMethod()) {
        // ...
    } elseif ($exception->payment->requiresConfirmation()) {
        // ...
    }
}
```

Strong Customer Authentication

If your business or one of your customers is based in Europe you will need to abide by the EU's Strong Customer Authentication (SCA) regulations. These regulations were imposed in September 2019 by the European Union to prevent payment fraud. Luckily, Stripe and Cashier are prepared for building SCA compliant applications.

{note} Before getting started, review Stripe's guide on PSD2 and SCA as well as their documentation on the new SCA APIs.

Payments Requiring Additional Confirmation

SCA regulations often require extra verification in order to confirm and process a payment. When this happens, Cashier will throw a `Laravel\Cashier\Exceptions\IncompletePayment` exception that informs you that extra verification is needed. More information on how to handle these exceptions be found can be found in the documentation on handling failed payments.

Payment confirmation screens presented by Stripe or Cashier may be tailored to a specific bank or card issuer's payment flow and can include additional card confirmation, a temporary small charge, separate device authentication, or other forms of verification.

Incomplete and Past Due State

When a payment needs additional confirmation, the subscription will remain in an `incomplete` or `past_due` state as indicated by its `stripe_status` database column. Cashier will automatically activate the customer's subscription as soon as payment confirmation is complete and your application is notified by Stripe via webhook of its completion.

For more information on `incomplete` and `past_due` states, please refer to our additional documentation on these states.

Off-Session Payment Notifications

Since SCA regulations require customers to occasionally verify their payment details even while their subscription is active, Cashier can send a notification to the customer when off-session payment confirmation is required. For example, this may occur when a subscription

is renewing. Cashier's payment notification can be enabled by setting the `CASHIER_PAYMENT_NOTIFICATION` environment variable to a notification class. By default, this notification is disabled. Of course, Cashier includes a notification class you may use for this purpose, but you are free to provide your own notification class if desired:

```
CASHIER_PAYMENT_NOTIFICATION=Laravel\Cashier\Notifications\ConfirmPaymen
t
```

To ensure that off-session payment confirmation notifications are delivered, verify that Stripe webhooks are configured for your application and the `invoice.payment_action_required` webhook is enabled in your Stripe dashboard. In addition, your `Billable` model should also use Laravel's `Illuminate\Notifications\Notifiable` trait.

{note} Notifications will be sent even when customers are manually making a payment that requires additional confirmation. Unfortunately, there is no way for Stripe to know that the payment was done manually or "off-session". But, a customer will simply see a "Payment Successful" message if they visit the payment page after already confirming their payment. The customer will not be allowed to accidentally confirm the same payment twice and incur an accidental second charge.

Stripe SDK

Many of Cashier's objects are wrappers around Stripe SDK objects. If you would like to interact with the Stripe objects directly, you may conveniently retrieve them using the `asStripe` method:

```
$stripeSubscription = $subscription->asStripeSubscription();

$stripeSubscription->application_fee_percent = 5;

$stripeSubscription->save();
```

You may also use the `updateStripeSubscription` method to update a Stripe subscription directly:

```
$subscription->updateStripeSubscription(['application_fee_percent' =>
5]);
```

You may invoke the `stripe` method on the `Cashier` class if you would like to use the `Stripe\StripeClient` client directly. For example, you could use this method to access the `StripeClient` instance and retrieve a list of prices from your Stripe account:

```
use Laravel\Cashier\Cashier;

$prices = Cashier::stripe()->prices->all();
```

Testing

When testing an application that uses Cashier, you may mock the actual HTTP requests to the Stripe API; however, this requires you to partially re-implement Cashier's own behavior. Therefore, we recommend allowing your tests to hit the actual Stripe API. While this is slower, it provides more confidence that your application is working as expected and any slow tests may be placed within their own PHPUnit testing group.

When testing, remember that Cashier itself already has a great test suite, so you should only focus on testing the subscription and payment flow of your own application and not every underlying Cashier behavior.

To get started, add the **testing** version of your Stripe secret to your `phpunit.xml` file:

```
<env name="STRIPE_SECRET" value="sk_test_<your-key>"/>
```

Now, whenever you interact with Cashier while testing, it will send actual API requests to your Stripe testing environment. For convenience, you should pre-fill your Stripe testing account with subscriptions / prices that you may use during testing.

{tip} In order to test a variety of billing scenarios, such as credit card denials and failures, you may use the vast range of testing card numbers and tokens provided by Stripe.

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. WebSockets provide a more efficient alternative to continually polling your application's server for data changes that should be reflected in your UI.

For example, imagine your application is able to export a user's data to a CSV file and email it to them. However, creating this CSV file takes several minutes so you choose to create and mail the CSV within a queued job. When the CSV has been created and mailed to the user, we can use event broadcasting to dispatch a `App\Events\UserDataExported` event that is received by our application's JavaScript. Once the event is received, we can display a message to the user that their CSV has been emailed to them without them ever needing to refresh the page.

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

The core concepts behind broadcasting are simple: clients connect to named channels on the frontend, while your Laravel application broadcasts events to these channels on the backend. These events can contain any additional data you wish to make available to the frontend.

Supported Drivers

By default, Laravel includes two server-side broadcasting drivers for you to choose from: Pusher Channels and Ably. However, community driven packages such as laravel-websockets provide additional broadcasting drivers that do not require commercial broadcasting providers.

{tip} Before diving into event broadcasting, make sure you have read Laravel's documentation on events and listeners.

Server Side Installation

To get started using Laravel's event broadcasting, we need to do some configuration within the Laravel application as well as install a few packages.

Event broadcasting is accomplished by a server-side broadcasting driver that broadcasts your Laravel events so that Laravel Echo (a JavaScript library) can receive them within the browser client. Don't worry - we'll walk through each part of the installation process step-by-step.

Configuration

All of your application's event broadcasting configuration is stored in the `config/broadcasting.php` configuration file. Laravel supports several broadcast drivers out of the box: Pusher Channels, Redis, and a `log` driver for local development and debugging. Additionally, a `null` driver is included which allows you to totally disable broadcasting during testing. A configuration example is included for each of these drivers in the `config/broadcasting.php` configuration file.

Broadcast Service Provider

Before broadcasting any events, you will first need to register the `App\Providers\BroadcastServiceProvider`. In new Laravel applications, you only need to uncomment this provider in the `providers` array of your `config/app.php` configuration file. This `BroadcastServiceProvider` contains the code necessary to register the broadcast authorization routes and callbacks.

Queue Configuration

You will also need to configure and run a queue worker. All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected by events being broadcast.

Pusher Channels

If you plan to broadcast your events using Pusher Channels, you should install the Pusher Channels PHP SDK using the Composer package manager:

```
composer require pusher/pusher-php-server
```

Next, you should configure your Pusher Channels credentials in the `config/broadcasting.php` configuration file. An example Pusher Channels configuration is already included in this file, allowing you to quickly specify your key, secret, and application ID. Typically, these values should be set via the `PUSHER_APP_KEY`, `PUSHER_APP_SECRET`, and `PUSHER_APP_ID` environment variables:

```
PUSHER_APP_ID=your-pusher-app-id
PUSHER_APP_KEY=your-pusher-key
PUSHER_APP_SECRET=your-pusher-secret
PUSHER_APP_CLUSTER=mt1
```

The `config/broadcasting.php` file's `pusher` configuration also allows you to specify additional `options` that are supported by Channels, such as the cluster.

Next, you will need to change your broadcast driver to `pusher` in your `.env` file:

```
BROADCAST_DRIVER=pusher
```

Finally, you are ready to install and configure Laravel Echo, which will receive the broadcast events on the client-side.

Pusher Compatible Laravel Websockets

The laravel-websockets package is a pure PHP, Pusher compatible WebSocket package for Laravel. This package allows you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using this package, please consult its official documentation.

Ably

If you plan to broadcast your events using Ably, you should install the Ably PHP SDK using the Composer package manager:

```
composer require ably/ably-php
```

Next, you should configure your Ably credentials in the `config/broadcasting.php` configuration file. An example Ably configuration is already included in this file, allowing you to quickly specify your key. Typically, this value should be set via the `ABLY_KEY` environment variable:

```
ABLY_KEY=your-ably-key
```

Next, you will need to change your broadcast driver to `ably` in your `.env` file:

```
BROADCAST_DRIVER=ably
```

Finally, you are ready to install and configure Laravel Echo, which will receive the broadcast events on the client-side.

Open Source Alternatives

The laravel-websockets package is a pure PHP, Pusher compatible WebSocket package for Laravel. This package allows you to leverage the full power of Laravel broadcasting without a commercial WebSocket provider. For more information on installing and using this package, please consult its official documentation.

Client Side Installation

Pusher Channels

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package since we will be using the Pusher Channels broadcaster:

```
npm install --save-dev laravel-echo pusher-js
```

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file - you simply need to uncomment it:

```
import Echo from 'laravel-echo';

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run dev
```

{tip} To learn more about compiling your application's JavaScript assets, please consult the documentation on Laravel Mix.

Using An Existing Client Instance

If you already have a pre-configured Pusher Channels client instance that you would like Echo to utilize, you may pass it to Echo via the `client` configuration option:

```
import Echo from 'laravel-echo';

const client = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    client: client
});
```

Ably

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by your server-side broadcasting driver. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package.

You may wonder why we would install the `pusher-js` JavaScript library even though we are using Ably to broadcast our events. Thankfully, Ably includes a Pusher compatibility mode which lets us use the Pusher protocol when listening for events in our client-side application:

```
npm install --save-dev laravel-echo pusher-js
```

Before continuing, you should enable Pusher protocol support in your Ably application settings. You may enable this feature within the "Protocol Adapter Settings" portion of your Ably application's settings dashboard.

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/js/bootstrap.js` file that is included with the Laravel framework. By default, an example Echo configuration is already included in this file; however, the default configuration in the `bootstrap.js` file is intended for Pusher. You may copy the configuration below to transition your configuration to Ably:

```
import Echo from 'laravel-echo';

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_ABLY_PUBLIC_KEY,
    wsHost: 'realtime-pusher.ably.io',
    wsPort: 443,
    disableStats: true,
    encrypted: true,
});
```

Note that our Ably Echo configuration references a `MIX_ABLY_PUBLIC_KEY` environment variable. This variable's value should be your Ably public key. Your public key is the portion of your Ably key that occurs before the `:` character.

Once you have uncommented and adjusted the Echo configuration according to your needs, you may compile your application's assets:

```
npm run dev
```

{tip} To learn more about compiling your application's JavaScript assets, please consult the documentation on Laravel Mix.

Concept Overview

Laravel's event broadcasting allows you to broadcast your server-side Laravel events to your client-side JavaScript application using a driver-based approach to WebSockets. Currently, Laravel ships with Pusher Channels and Ably drivers. The events may be easily consumed on the client-side using the Laravel Echo JavaScript package.

Events are broadcast over "channels", which may be specified as public or private. Any visitor to your application may subscribe to a public channel without any authentication or authorization; however, in order to subscribe to a private channel, a user must be authenticated and authorized to listen on that channel.

{tip} If you would like to use an open source, PHP driven alternative to Pusher, check out the laravel-websockets package.

Using An Example Application

Before diving into each component of event broadcasting, let's take a high level overview using an e-commerce store as an example.

In our application, let's assume we have a page that allows users to view the shipping status for their orders. Let's also assume that a `OrderShipmentStatusUpdated` event is fired when a shipping status update is processed by the application:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

The `ShouldBroadcast` Interface

When a user is viewing one of their orders, we don't want them to have to refresh the page to view status updates. Instead, we want to broadcast the updates to the application as they are created. So, we need to mark the `OrderShipmentStatusUpdated` event with the `ShouldBroadcast` interface. This will instruct Laravel to broadcast the event when it is fired:

```
<?php

namespace App\Events;

use App\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * The order instance.
     *
     * @var \App\Order
     */
    public $order;
}
```

The `ShouldBroadcast` interface requires our event to define a `broadcastOn` method. This method is responsible for returning the channels that the event should broadcast on. An empty stub of this method is already defined on generated event classes, so we only need to fill in its details. We only want the creator of the order to be able to view status updates, so we will broadcast the event on a private channel that is tied to the order:

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return \Illuminate\Broadcasting\PrivateChannel
 */
public function broadcastOn()
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

Authorizing Channels

Remember, users must be authorized to listen on private channels. We may define our

channel authorization rules in our application's `routes/channels.php` file. In this example, we need to verify that any user attempting to listen on the private `order.1` channel is actually the creator of the order:

```
use App\Models\Order;

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

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Listening For Event Broadcasts

Next, all that remains is to listen for the event in our JavaScript application. We can do this using Laravel Echo. First, we'll use the `private` method to subscribe to the private channel. Then, we may use the `listen` method to listen for the `OrderShipmentStatusUpdated` event. By default, all of the event's public properties will be included on the broadcast event:

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
    });
```

Defining Broadcast Events

To inform Laravel that a given event should be broadcast, you must implement the `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface on the event class. This interface is already imported into all event classes generated by the framework so you may easily add it to any of your events.

The `ShouldBroadcast` interface requires you to implement a single method: `broadcastOn`. The `broadcastOn` method should return a channel or array of channels that the event should broadcast on. The channels should be instances of `Channel`, `PrivateChannel`, or `PresenceChannel`. Instances of `Channel` represent public channels that any user may subscribe to, while `PrivateChannels` and `PresenceChannels` represent private channels that require channel authorization:

```
<?php

namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * The user that created the server.
     *
     * @var \App\Models\User
     */
    public $user;

    /**
     * Create a new event instance.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
```

```
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('user.'.$this->user->id);
    }
}
```

After implementing the `ShouldBroadcast` interface, you only need to fire the event as you normally would. Once the event has been fired, a queued job will automatically broadcast the event using your specified broadcast driver.

Broadcast Name

By default, Laravel will broadcast the event using the event's class name. However, you may customize the broadcast name by defining a `broadcastAs` method on the event:

```
/**
 * The event's broadcast name.
 *
 * @return string
 */
public function broadcastAs()
{
    return 'server.created';
}
```

If you customize the broadcast name using the `broadcastAs` method, you should make sure to register your listener with a leading `.` character. This will instruct Echo to not prepend the application's namespace to the event:

```
.listen('.server.created', function (e) {
    ....
});
```

Broadcast Data

When an event is broadcast, all of its `public` properties are automatically serialized and broadcast as the event's payload, allowing you to access any of its public data from your JavaScript application. So, for example, if your event has a single public `$user` property that contains an Eloquent model, the event's broadcast payload would be:

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

However, if you wish to have more fine-grained control over your broadcast payload, you may add a `broadcastWith` method to your event. This method should return the array of data that you wish to broadcast as the event payload:

```
/**
 * Get the data to broadcast.
 *
 * @return array
 */
public function broadcastWith()
{
    return ['id' => $this->user->id];
}
```

Broadcast Queue

By default, each broadcast event is placed on the default queue for the default queue connection specified in your `queue.php` configuration file. You may customize the queue connection and name used by the broadcaster by defining `connection` and `queue` properties on your event class:

```
/**
 * The name of the queue connection to use when broadcasting the event.
 *
 * @var string
 */
public $connection = 'redis';

/**
 * The name of the queue on which to place the broadcasting job.
 *
 * @var string
 */
public $queue = 'default';
```

If you want to broadcast your event using the `sync` queue instead of the default queue driver, you can implement the `ShouldBroadcastNow` interface instead of `ShouldBroadcast`:

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class OrderShipmentStatusUpdated implements ShouldBroadcastNow
{
    //
}
```

Broadcast Conditions

Sometimes you want to broadcast your event only if a given condition is true. You may define these conditions by adding a `broadcastWhen` method to your event class:

```
/**
 * Determine if this event should broadcast.
 *
 * @return bool
 */
public function broadcastWhen()
{
    return $this->order->value > 100;
}
```

Broadcasting & Database Transactions

When broadcast events are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your event depends on these models, unexpected errors can occur when the job that broadcasts the event is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular broadcast event should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the event class:

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $afterCommit = true;
}
```

{tip} To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Authorizing Channels

Private channels require you to authorize that the currently authenticated user can actually listen on the channel. This is accomplished by making an HTTP request to your Laravel application with the channel name and allowing your application to determine if the user can listen on that channel. When using Laravel Echo, the HTTP request to authorize subscriptions to private channels will be made automatically; however, you do need to define the proper routes to respond to these requests.

Defining Authorization Routes

Thankfully, Laravel makes it easy to define the routes to respond to channel authorization requests. In the `App\Providers\BroadcastServiceProvider` included with your Laravel application, you will see a call to the `Broadcast::routes` method. This method will register the `/broadcasting/auth` route to handle authorization requests:

```
Broadcast::routes();
```

The `Broadcast::routes` method will automatically place its routes within the `web` middleware group; however, you may pass an array of route attributes to the method if you would like to customize the assigned attributes:

```
Broadcast::routes($attributes);
```

Customizing The Authorization Endpoint

By default, Echo will use the `/broadcasting/auth` endpoint to authorize channel access. However, you may specify your own authorization endpoint by passing the `authEndpoint` configuration option to your Echo instance:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    authEndpoint: '/custom/endpoint/auth'
});
```

Customizing The Authorization Request

You can customize how Laravel Echo performs authorization requests by providing a custom authorizer when initializing Echo:

```
window.Echo = new Echo({
    // ...
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(false, response.data);
                })
                .catch(error => {
                    callback(true, error);
                });
            }
        };
    },
})
```

Defining Authorization Callbacks

Next, we need to define the logic that will actually determine if the currently authenticated user can listen to a given channel. This is done in the `routes/channels.php` file that is included with your application. In this file, you may use the `Broadcast::channel` method to register channel authorization callbacks:

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

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Authorization Callback Model Binding

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

```
use App\Models\Order;

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

{note} Unlike HTTP route model binding, channel model binding does not support automatic implicit model binding scoping. However, this is rarely a problem because most channels can be scoped based on a single model's unique, primary key.

Authorization Callback Authentication

Private and presence broadcast channels authenticate the current user via your application's default authentication guard. If the user is not authenticated, channel authorization is automatically denied and the authorization callback is never executed. However, you may assign multiple, custom guards that should authenticate the incoming request if necessary:

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]);
```

Defining Channel Classes

If your application is consuming many different channels, your `routes/channels.php` file could become bulky. So, instead of using closures to authorize channels, you may use channel classes. To generate a channel class, use the `make:channel` Artisan command. This command will place a new channel class in the `App/Broadcasting` directory.

```
php artisan make:channel OrderChannel
```

Next, register your channel in your `routes/channels.php` file:

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

Finally, you may place the authorization logic for your channel in the channel class' `join` method. This `join` method will house the same logic you would have typically placed in your channel authorization closure. You may also take advantage of channel model binding:

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * Create a new channel instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Authenticate the user's access to the channel.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Order  $order
     * @return array|bool
     */
    public function join(User $user, Order $order)
    {
        return $user->id === $order->user_id;
    }
}
```

{tip} Like many other classes in Laravel, channel classes will automatically be resolved by the service container. So, you may type-hint any dependencies required by your channel in its constructor.

Broadcasting Events

Once you have defined an event and marked it with the `ShouldBroadcast` interface, you only need to fire the event using the event's dispatch method. The event dispatcher will notice that the event is marked with the `ShouldBroadcast` interface and will queue the event for broadcasting:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

Only To Others

When building an application that utilizes event broadcasting, you may occasionally need to broadcast an event to all subscribers to a given channel except for the current user. You may accomplish this using the `broadcast` helper and the `toOthers` method:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

To better understand when you may want to use the `toOthers` method, let's imagine a task list application where a user may create a new task by entering a task name. To create a task, your application might make a request to a `/task` URL which broadcasts the task's creation and returns a JSON representation of the new task. When your JavaScript application receives the response from the end-point, it might directly insert the new task into its task list like so:

```
axios.post('/task', task)
    .then((response) => {
        this.tasks.push(response.data);
    });
```

However, remember that we also broadcast the task's creation. If your JavaScript application is also listening for this event in order to add tasks to the task list, you will have duplicate

tasks in your list: one from the end-point and one from the broadcast. You may solve this by using the `toOthers` method to instruct the broadcaster to not broadcast the event to the current user.

{note} Your event must use the `Illuminate\Broadcasting\InteractsWithSockets` trait in order to call the `toOthers` method.

Configuration

When you initialize a Laravel Echo instance, a socket ID is assigned to the connection. If you are using a global Axios instance to make HTTP requests from your JavaScript application, the socket ID will automatically be attached to every outgoing request as a `X-Socket-ID` header. Then, when you call the `toOthers` method, Laravel will extract the socket ID from the header and instruct the broadcaster to not broadcast to any connections with that socket ID.

If you are not using a global Axios instance, you will need to manually configure your JavaScript application to send the `X-Socket-ID` header with all outgoing requests. You may retrieve the socket ID using the `Echo.socketId` method:

```
var socketId = Echo.socketId();
```

Customizing The Connection

If your application interacts with multiple broadcast connections and you want to broadcast an event using a broadcaster other than your default, you may specify which connection to push an event to using the `via` method:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->via('pusher');
```

Alternatively, you may specify the event's broadcast connection by calling the `broadcastVia` method within the event's constructor. However, before doing so, you should ensure that the event class uses the `InteractsWithBroadcasting` trait:

```php
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithBroadcasting;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    use InteractsWithBroadcasting;
    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->broadcastVia('pusher');
    }
}
```

Receiving Broadcasts

Listening For Events

Once you have installed and instantiated Laravel Echo, you are ready to start listening for events that are broadcast from your Laravel application. First, use the `channel` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event:

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
    });
```

If you would like to listen for events on a private channel, use the `private` method instead. You may continue to chain calls to the `listen` method to listen for multiple events on a single channel:

```
Echo.private(`orders.${this.order.id}`)
    .listen(...)
    .listen(...)
    .listen(...);
```

Stop Listening For Events

If you would like to stop listening to a given event without leaving the channel, you may use the `stopListening` method:

```
Echo.private(`orders.${this.order.id}`)
    .stopListening('OrderShipmentStatusUpdated')
```

Leaving A Channel

To leave a channel, you may call the `leaveChannel` method on your Echo instance:

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

If you would like to leave a channel and also its associated private and presence channels, you may call the `leave` method:

```
Echo.leave(`orders.${this.order.id}`);
```

Namespaces

You may have noticed in the examples above that we did not specify the full `App\Events` namespace for the event classes. This is because Echo will automatically assume the events are located in the `App\Events` namespace. However, you may configure the root namespace when you instantiate Echo by passing a `namespace` configuration option:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    namespace: 'App.Other.Namespace'
});
```

Alternatively, you may prefix event classes with a `.` when subscribing to them using Echo. This will allow you to always specify the fully-qualified class name:

```
Echo.channel('orders')
    .listen('.Namespace\\Event\\Class', (e) => {
        //
    });
```

Presence Channels

Presence channels build on the security of private channels while exposing the additional feature of awareness of who is subscribed to the channel. This makes it easy to build powerful, collaborative application features such as notifying users when another user is viewing the same page or listing the inhabitants of a chat room.

Authorizing Presence Channels

All presence channels are also private channels; therefore, users must be authorized to access them. However, when defining authorization callbacks for presence channels, you will not return `true` if the user is authorized to join the channel. Instead, you should return an array of data about the user.

The data returned by the authorization callback will be made available to the presence channel event listeners in your JavaScript application. If the user is not authorized to join the presence channel, you should return `false` or `null`:

```
Broadcast::channel('chat.{roomId}', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

Joining Presence Channels

To join a presence channel, you may use Echo's `join` method. The `join` method will return a `PresenceChannel` implementation which, along with exposing the `listen` method, allows you to subscribe to the `here`, `joining`, and `leaving` events.

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        //
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    })
    .error((error) => {
        console.error(error);
    });
```

The `here` callback will be executed immediately once the channel is joined successfully, and will receive an array containing the user information for all of the other users currently subscribed to the channel. The `joining` method will be executed when a new user joins a channel, while the `leaving` method will be executed when a user leaves the channel. The `error` method will be executed when the authentication endpoint returns a HTTP status code other than 200 or if there is a problem parsing the returned JSON.

Broadcasting To Presence Channels

Presence channels may receive events just like public or private channels. Using the example of a chatroom, we may want to broadcast `NewMessage` events to the room's presence channel. To do so, we'll return an instance of `PresenceChannel` from the event's `broadcastOn` method:

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return Channel|array
 */
public function broadcastOn()
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

As with other events, you may use the `broadcast` helper and the `toOthers` method to exclude the current user from receiving the broadcast:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

As typical of other types of events, you may listen for events sent to presence channels using Echo's `listen` method:

```
Echo.join(`chat.${roomId}`)
    .here(...)
    .joining(...)
    .leaving(...)
    .listen('NewMessage', (e) => {
        //
    });
```

Model Broadcasting

{note} Before reading the following documentation about model broadcasting, we recommend you become familiar with the general concepts of Laravel's model broadcasting services as well as how to manually create and listen to broadcast events.

It is common to broadcast events when your application's Eloquent models are created, updated, or deleted. Of course, this can easily be accomplished by manually defining custom events for Eloquent model state changes and marking those events with the `ShouldBroadcast` interface.

However, if you are not using these events for any other purposes in your application, it can be cumbersome to create event classes for the sole purpose of broadcasting them. To remedy this, Laravel allows you to indicate that an Eloquent model should automatically broadcast its state changes.

To get started, your Eloquent model should use the `Illuminate\Database\Eloquent\BroadcastsEvents` trait. In addition, the model should define a `broadcastsOn` method, which will return an array of channels that the model's events should broadcast on:

```php
<?php

namespace App\Models;

use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use BroadcastsEvents, HasFactory;

    /**
     * Get the user that the post belongs to.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }

    /**
     * Get the channels that model events should broadcast on.
     *
     * @param  string  $event
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn($event)
    {
        return [$this, $this->user];
    }
}
```

Once your model includes this trait and defines its broadcast channels, it will begin automatically broadcasting events when a model instance is created, updated, deleted, trashed, or restored.

In addition, you may have noticed that the `broadcastOn` method receives a string `$event` argument. This argument contains the type of event that has occurred on the model and will have a value of `created`, `updated`, `deleted`, `trashed`, or `restored`. By inspecting the value of this variable, you may determine which channels (if any) the model should broadcast to for a particular event:

```php
/**
 * Get the channels that model events should broadcast on.
 *
 * @param  string  $event
 * @return \Illuminate\Broadcasting\Channel|array
 */
public function broadcastOn($event)
{
    return match ($event) {
        'deleted' => [],
        default => [$this, $this->user],
    };
}
```

Customizing Model Broadcasting Event Creation

Occasionally, you may wish to customize how Laravel creates the underlying model broadcasting event. You may accomplish this by defining a `newBroadcastableEvent` method on your Eloquent model. This method should return an `Illuminate\Database\Eloquent\BroadcastableModelEventOccurred` instance:

```php
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred

/**
 * Create a new broadcastable model event for the model.
 *
 * @param  string  $event
 * @return \Illuminate\Database\Eloquent\BroadcastableModelEventOccurred
 */
protected function newBroadcastableEvent($event)
{
    return (new BroadcastableModelEventOccurred(
        $this, $event
    ))->dontBroadcastToCurrentUser();
}
```

Model Broadcasting Conventions

Channel Conventions

As you may have noticed, the `broadcastOn` method in the model example above did not return `Channel` instances. Instead, Eloquent models were returned directly. If an Eloquent model instance is returned by your model's `broadcastOn` method (or is contained in an array returned by the method), Laravel will automatically instantiate a private channel instance for the model using the model's class name and primary key identifier as the channel name.

So, an `App\Models\User` model with an `id` of `1` would be converted into a `Illuminate\Broadcasting\PrivateChannel` instance with a name of `App.Models.User.1`. Of course, in addition to returning Eloquent model instances from your model's `broadcastOn` method, you may return complete `Channel` instances in order to have full control over the model's channel names:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channels that model events should broadcast on.
 *
 * @param  string  $event
 * @return \Illuminate\Broadcasting\Channel|array
 */
public function broadcastOn($event)
{
    return [new PrivateChannel('user.'.$this->id)];
}
```

If you plan to explicitly return a channel instance from your model's `broadcastOn` method, you may pass an Eloquent model instance to the channel's constructor. When doing so, Laravel will use the model channel conventions discussed above to convert the Eloquent model into a channel name string:

```
return [new Channel($this->user)];
```

If you need to determine the channel name of a model, you may call the `broadcastChannel` method on any model instance. For example, this method returns the string `App.Models.User.1` for a `App\Models\User` model with an `id` of `1`:

```
$user->broadcastChannel()
```

Event Conventions

Since model broadcast events are not associated with an "actual" event within your application's `App\Events` directory, they are assigned a name and a payload based on conventions. Laravel's convention is to broadcast the event using the class name of the model (not including the namespace) and the name of the model event that triggered the broadcast.

So, for example, an update to the `App\Models\Post` model would broadcast an event to your client-side application as `PostUpdated` with the following payload:

```
{
    "model": {
        "id": 1,
        "title": "My first post"
        ...
    },
    ...
    "socket": "someSocketId",
}
```

The deletion of the `App\Models\User` model would broadcast an event named `UserDeleted`.

If you would like, you may define a custom broadcast name and payload by adding a `broadcastAs` and `broadcastWith` method to your model. These methods receive the name of the model event / operation that is occurring, allowing you to customize the event's name and payload for each model operation. If `null` is returned from the `broadcastAs` method, Laravel will use the model broadcasting event name conventions discussed above when broadcasting the event:

```php
/**
 * The model event's broadcast name.
 *
 * @param  string  $event
 * @return string|null
 */
public function broadcastAs($event)
{
    return match ($event) {
        'created' => 'post.created',
        default => null,
    };
}

/**
 * Get the data to broadcast for the model.
 *
 * @param  string  $event
 * @return array
 */
public function broadcastWith($event)
{
    return match ($event) {
        'created' => ['title' => $this->title],
        default => ['model' => $this],
    };
}
```

Listening For Model Broadcasts

Once you have added the `BroadcastsEvents` trait to your model and defined your model's `broadcastOn` method, you are ready to start listening for broadcasted model events within your client-side application. Before getting started, you may wish to consult the complete documentation on listening for events.

First, use the `private` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event. Typically, the channel name given to the `private` method should correspond to Laravel's model broadcasting conventions.

Once you have obtained a channel instance, you may use the `listen` method to listen for a particular event. Since model broadcast events are not associated with an "actual" event within your application's `App\Events` directory, the event name must be prefixed with a `.`

to indicate it does not belong to a particular namespace. Each model broadcast event has a `model` property which contains all of the broadcastable properties of the model:

```
Echo.private(`App.Models.User.${this.user.id}`)
    .listen('.PostUpdated', (e) => {
        console.log(e.model);
    });
```

Client Events

{tip} When using Pusher Channels, you must enable the "Client Events" option in the "App Settings" section of your application dashboard in order to send client events.

Sometimes you may wish to broadcast an event to other connected clients without hitting your Laravel application at all. This can be particularly useful for things like "typing" notifications, where you want to alert users of your application that another user is typing a message on a given screen.

To broadcast client events, you may use Echo's `whisper` method:

```
Echo.private(`chat.${roomId}`)
    .whisper('typing', {
        name: this.user.name
    });
```

To listen for client events, you may use the `listenForWhisper` method:

```
Echo.private(`chat.${roomId}`)
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

Notifications

By pairing event broadcasting with notifications, your JavaScript application may receive new notifications as they occur without needing to refresh the page. Before getting started, be sure to read over the documentation on using the broadcast notification channel.

Once you have configured a notification to use the broadcast channel, you may listen for the broadcast events using Echo's `notification` method. Remember, the channel name should match the class name of the entity receiving the notifications:

```
Echo.private(`App.Models.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
    });
```

In this example, all notifications sent to `App\Models\User` instances via the `broadcast` channel would be received by the callback. A channel authorization callback for the `App.Models.User.{id}` channel is included in the default `BroadcastServiceProvider` that ships with the Laravel framework.

Laravel Cashier (Paddle)

Introduction

Laravel Cashier Paddle provides an expressive, fluent interface to Paddle's subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading. In addition to basic subscription management, Cashier can handle: coupons, swapping subscription, subscription "quantities", cancellation grace periods, and more.

While working with Cashier we recommend you also review Paddle's user guides and API documentation.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review the upgrade guide.

Installation

First, install the Cashier package for Paddle using the Composer package manager:

```
composer require laravel/cashier-paddle
```

{note} To ensure Cashier properly handles all Paddle events, remember to set up Cashier's webhook handling.

Paddle Sandbox

During local and staging development, you should register a Paddle Sandbox account. This account will give you a sandboxed environment to test and develop your applications without making actual payments. You may use Paddle's test card numbers to simulate various payment scenarios.

After you have finished developing your application you may apply for a Paddle vendor account.

Database Migrations

The Cashier service provider registers its own database migration directory, so remember to migrate your database after installing the package. The Cashier migrations will create a new `customers` table. In addition, a new `subscriptions` table will be created to store all of your customer's subscriptions. Finally, a new `receipts` table will be created to store all of your application's receipt information:

```
php artisan migrate
```

If you need to overwrite the migrations that are included with Cashier, you can publish them using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

If you would like to prevent Cashier's migrations from running entirely, you may use the `ignoreMigrations` provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Paddle\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::ignoreMigrations();
}
```

Configuration

Billable Model

Before using Cashier, you must add the `Billable` trait to your user model definition. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons and updating payment method information:

```
use Laravel\Paddle\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

If you have billable entities that are not users, you may also add the trait to those classes:

```
use Illuminate\Database\Eloquent\Model;
use Laravel\Paddle\Billable;

class Team extends Model
{
    use Billable;
}
```

API Keys

Next, you should configure your Paddle keys in your application's `.env` file. You can retrieve your Paddle API keys from the Paddle control panel:

```
PADDLE_VENDOR_ID=your-paddle-vendor-id
PADDLE_VENDOR_AUTH_CODE=your-paddle-vendor-auth-code
PADDLE_PUBLIC_KEY="your-paddle-public-key"
PADDLE_SANDBOX=true
```

The `PADDLE_SANDBOX` environment variable should be set to `true` when you are using Paddle's Sandbox environment. The `PADDLE_SANDBOX` variable should be set to `false` if you are deploying your application to production and are using Paddle's live vendor environment.

Paddle JS

Paddle relies on its own JavaScript library to initiate the Paddle checkout widget. You can load the JavaScript library by placing the `@paddleJS` Blade directive right before your application layout's closing `</head>` tag:

```
<head>
    ...

    @paddleJS
</head>
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by defining a `CASHIER_CURRENCY` environment variable within your application's `.env` file:

```
CASHIER_CURRENCY=EUR
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's `NumberFormatter` class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```

{note} In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Overriding Default Models

You are free to extend the models used internally by Cashier by defining your own model and extending the corresponding Cashier model:

```
use Laravel\Paddle\Subscription as CashierSubscription;

class Subscription extends CashierSubscription
{
    // ...
}
```

After defining your model, you may instruct Cashier to use your custom model via the `Laravel\Paddle\Cashier` class. Typically, you should inform Cashier about your custom models in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use App\Models\Cashier\Receipt;
use App\Models\Cashier\Subscription;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Cashier::useReceiptModel(Receipt::class);
    Cashier::useSubscriptionModel(Subscription::class);
}
```

Core Concepts

Pay Links

Paddle lacks an extensive CRUD API to perform subscription state changes. Therefore, most interactions with Paddle are done through its checkout widget. Before we can display the checkout widget, we must generate a "pay link" using Cashier. A "pay link" will inform the checkout widget of the billing operation we wish to perform:

```
use App\Models\User;
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $request->user()->newSubscription('default', $premium =
34567)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

Cashier includes a `paddle-button` Blade component. We may pass the pay link URL to this component as a "prop". When this button is clicked, Paddle's checkout widget will be displayed:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

By default, this will display a button with the standard Paddle styling. You can remove all Paddle styling by adding the `data-theme="none"` attribute to the component:

```
<x-paddle-button :url="$payLink" class="px-8 py-4" data-theme="none">
    Subscribe
</x-paddle-button>
```

The Paddle checkout widget is asynchronous. Once the user creates or updates a subscription within the widget, Paddle will send your application webhooks so that you may properly update the subscription state in our own database. Therefore, it's important that you properly set up webhooks to accommodate for state changes from Paddle.

For more information on pay links, you may review the Paddle API documentation on pay link generation.

{note} After a subscription state change, the delay for receiving the corresponding webhook is typically minimal but you should account for this in your application by considering that your user's subscription might not be immediately available after completing the checkout.

Manually Rendering Pay Links

You may also manually render a pay link without using Laravel's built-in Blade components. To get started, generate the pay link URL as demonstrated in previous examples:

```
$payLink = $request->user()->newSubscription('default', $premium =
34567)
    ->returnTo(route('home'))
    ->create();
```

Next, simply attach the pay link URL to an a element in your HTML:

```
<a href="#!" class="ml-4 paddle_button" data-override="{{ $payLink }}">
    Paddle Checkout
</a>
```

Payments Requiring Additional Confirmation

Sometimes additional verification is required in order to confirm and process a payment. When this happens, Paddle will present a payment confirmation screen. Payment confirmation screens presented by Paddle or Cashier may be tailored to a specific bank or card issuer's payment flow and can include additional card confirmation, a temporary small

charge, separate device authentication, or other forms of verification.

Inline Checkout

If you don't want to make use of Paddle's "overlay" style checkout widget, Paddle also provides the option to display the widget inline. While this approach does not allow you to adjust any of the checkout's HTML fields, it allows you to embed the widget within your application.

To make it easy for you to get started with inline checkout, Cashier includes a `paddle-checkout` Blade component. To get started, you should generate a pay link and pass the pay link to the component's `override` attribute:

```
<x-paddle-checkout :override="$payLink" class="w-full" />
```

To adjust the height of the inline checkout component, you may pass the `height` attribute to the Blade component:

```
<x-paddle-checkout :override="$payLink" class="w-full" height="500" />
```

Inline Checkout Without Pay Links

Alternatively, you may customize the widget with custom options instead of using a pay link:

```
$options = [
    'product' => $productId,
    'title' => 'Product Title',
];

<x-paddle-checkout :options="$options" class="w-full" />
```

Please consult Paddle's guide on Inline Checkout as well as their parameter reference for further details on the inline checkout's available options.

{note} If you would like to also use the `passthrough` option when specifying custom options, you should provide a key / value array as its value. Cashier will automatically handle converting the array to a JSON string. In addition, the `customer_id` passthrough option is reserved for internal Cashier usage.

Manually Rendering An Inline Checkout

You may also manually render an inline checkout without using Laravel's built-in Blade components. To get started, generate the pay link URL as demonstrated in previous examples.

Next, you may use Paddle.js to initialize the checkout. To keep this example simple, we will demonstrate this using Alpine.js; however, you are free to translate this example to your own frontend stack:

```
<div class="paddle-checkout" x-data="{}" x-init="
    Paddle.Checkout.open({
        override: {{ $payLink }},
        method: 'inline',
        frameTarget: 'paddle-checkout',
        frameInitialHeight: 366,
        frameStyle: 'width: 100%; background-color: transparent; border:
none;'
    });
">
</div>
```

User Identification

In contrast to Stripe, Paddle users are unique across all of Paddle, not unique per Paddle account. Because of this, Paddle's API's do not currently provide a method to update a user's details such as their email address. When generating pay links, Paddle identifies users using the `customer_email` parameter. When creating a subscription, Paddle will try to match the user provided email to an existing Paddle user.

In light of this behavior, there are some important things to keep in mind when using Cashier and Paddle. First, you should be aware that even though subscriptions in Cashier are tied to the same application user, **they could be tied to different users within Paddle's internal systems**. Secondly, each subscription has its own connected payment method information and could also have different email addresses within Paddle's internal systems (depending on which email was assigned to the user when the subscription was created).

Therefore, when displaying subscriptions you should always inform the user which email address or payment method information is connected to the subscription on a per-subscription basis. Retrieving this information can be done with the following methods provided by the `Laravel\Paddle\Subscription` model:

```
$subscription = $user->subscription('default');

$subscription->paddleEmail();
$subscription->paymentMethod();
$subscription->cardBrand();
$subscription->cardLastFour();
$subscription->cardExpirationDate();
```

There is currently no way to modify a user's email address through the Paddle API. When a user wants to update their email address within Paddle, the only way for them to do so is to contact Paddle customer support. When communicating with Paddle, they need to provide the `paddleEmail` value of the subscription to assist Paddle in updating the correct user.

Prices

Paddle allows you to customize prices per currency, essentially allowing you to configure different prices for different countries. Cashier Paddle allows you to retrieve all of the prices for a given product using the `productPrices` method. This method accepts the product IDs of the products you wish to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456]);
```

The currency will be determined based on the IP address of the request; however, you may optionally provide a specific country to retrieve prices for:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456], ['customer_country' =>
'BE']);
```

After retrieving the prices you may display them however you wish:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->gross()
}}</li>
    @endforeach
</ul>
```

You may also display the net price (excludes tax) and display the tax amount separately:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->net() }}
(+ {{ $price->price()->tax() }} tax)</li>
    @endforeach
</ul>
```

If you retrieved prices for subscription plans you can display their initial and recurring price separately:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - Initial: {{
$price->initialPrice()->gross() }} - Recurring: {{
$price->recurringPrice()->gross() }}</li>
    @endforeach
</ul>
```

For more information, check Paddle's API documentation on prices.

Customers

If a user is already a customer and you would like to display the prices that apply to that customer, you may do so by retrieving the prices directly from the customer instance:

```
use App\Models\User;

$prices = User::find(1)->productPrices([123, 456]);
```

Internally, Cashier will use the user's `paddleCountry` method to retrieve the prices in their currency. So, for example, a user living in the United States will see prices in USD while a user in Belgium will see prices in EUR. If no matching currency can be found the default currency of the product will be used. You can customize all prices of a product or subscription plan in the Paddle control panel.

Coupons

You may also choose to display prices after a coupon reduction. When calling the `productPrices` method, coupons may be passed as a comma delimited string:

```
use Laravel\Paddle\Cashier;

$prices = Cashier::productPrices([123, 456], [
    'coupons' => 'SUMMERSALE,20PERCENTOFF'
]);
```

Then, display the calculated prices using the `price` method:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{ $price->price()->gross()
}}</li>
    @endforeach
</ul>
```

You may display the original listed prices (without coupon discounts) using the `listPrice` method:

```
<ul>
    @foreach ($prices as $price)
        <li>{{ $price->product_title }} - {{
$price->listPrice()->gross() }}</li>
    @endforeach
</ul>
```

{note} When using the prices API, Paddle only allows applying coupons to one-time purchase products and not to subscription plans.

Customers

Customer Defaults

Cashier allows you to define some useful defaults for your customers when creating pay links. Setting these defaults allow you to pre-fill a customer's email address, country, and postal code so that they can immediately move on to the payment portion of the checkout widget. You can set these defaults by overriding the following methods on your billable model:

```
/**
 * Get the customer's email address to associate with Paddle.
 *
 * @return string|null
 */
public function paddleEmail()
{
    return $this->email;
}

/**
 * Get the customer's country to associate with Paddle.
 *
 * This needs to be a 2 letter code. See the link below for supported
countries.
 *
 * @return string|null
 * @link
https://developer.paddle.com/reference/platform-parameters/supported-cou
ntries
 */
public function paddleCountry()
{
    //
}

/**
 * Get the customer's postal code to associate with Paddle.
 *
 * See the link below for countries which require this.
 *
 * @return string|null
 * @link
https://developer.paddle.com/reference/platform-parameters/supported-cou
ntries#countries-requiring-postcode
 */
public function paddlePostcode()
{
    //
}
```

These defaults will be used for every action in Cashier that generates a pay link.

Subscriptions

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\Models\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription pay link:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $user->newSubscription('default', $premium = 12345)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

The first argument passed to the `newSubscription` method should be the internal name of the subscription. If your application only offers a single subscription, you might call this `default` or `primary`. This subscription name is only for internal application usage and is not meant to be shown to users. In addition, it should not contain spaces and it should never be changed after creating the subscription. The second argument given to the `newSubscription` method is the specific plan the user is subscribing to. This value should correspond to the plan's identifier in Paddle. The `returnTo` method accepts a URL that your user will be redirected to after they successfully complete the checkout.

The `create` method will create a pay link which you can use to generate a payment button. The payment button can be generated using the `paddle-button` Blade component that is included with Cashier Paddle:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Subscribe
</x-paddle-button>
```

After the user has finished their checkout, a `subscription_created` webhook will be dispatched from Paddle. Cashier will receive this webhook and setup the subscription for

your customer. In order to make sure all webhooks are properly received and handled by your application, ensure you have properly setup webhook handling.

Additional Details

If you would like to specify additional customer or subscription details, you may do so by passing them as an array of key / value pairs to the `create` method. To learn more about the additional fields supported by Paddle, check out Paddle's documentation on generating pay links:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->create([
        'vat_number' => $vatNumber,
    ]);
```

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->withCoupon('code')
    ->create();
```

Metadata

You can also pass an array of metadata using the `withMetadata` method:

```
$payLink = $user->newSubscription('default', $monthly = 12345)
    ->returnTo(route('home'))
    ->withMetadata(['key' => 'value'])
    ->create();
```

{note} When providing metadata, please avoid using `subscription_name` as a metadata key. This key is reserved for internal use by Cashier.

Checking Subscription Status

Once a user is subscribed to your application, you may check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the user has an active subscription, even if the subscription is currently within its trial period:

```
if ($user->subscribed('default')) {
    //
}
```

The `subscribed` method also makes a great candidate for a route middleware, allowing you to filter access to routes and controllers based on the user's subscription status:

```
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureUserIsSubscribed
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->user() && !
$request->user()->subscribed('default')) {
            // This user is not a paying customer...
            return redirect('billing');
        }

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

If you would like to determine if a user is still within their trial period, you may use the

`onTrial` method. This method can be useful for determining if you should display a warning to the user that they are still on their trial period:

```
if ($user->subscription('default')->onTrial()) {
    //
}
```

The `subscribedToPlan` method may be used to determine if the user is subscribed to a given plan based on a given Paddle plan ID. In this example, we will determine if the user's `default` subscription is actively subscribed to the monthly plan:

```
if ($user->subscribedToPlan($monthly = 12345, 'default')) {
    //
}
```

By passing an array to the `subscribedToPlan` method, you may determine if the user's `default` subscription is actively subscribed to the monthly or the yearly plan:

```
if ($user->subscribedToPlan([$monthly = 12345, $yearly = 54321],
'default')) {
    //
}
```

The `recurring` method may be used to determine if the user is currently subscribed and is no longer within their trial period:

```
if ($user->subscription('default')->recurring()) {
    //
}
```

Cancelled Subscription Status

To determine if the user was once an active subscriber but has cancelled their subscription, you may use the `cancelled` method:

```
if ($user->subscription('default')->cancelled()) {
    //
}
```

You may also determine if a user has cancelled their subscription, but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. Note that the `subscribed` method still returns `true` during this time:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

To determine if the user has cancelled their subscription and is no longer within their "grace period", you may use the `ended` method:

```
if ($user->subscription('default')->ended()) {
    //
}
```

Past Due Status

If a payment fails for a subscription, it will be marked as `past_due`. When your subscription is in this state it will not be active until the customer has updated their payment information. You may determine if a subscription is past due using the `pastDue` method on the subscription instance:

```
if ($user->subscription('default')->pastDue()) {
    //
}
```

When a subscription is past due, you should instruct the user to update their payment information. You may configure how past due subscriptions are handled in your Paddle subscription settings.

If you would like subscriptions to still be considered active when they are `past_due`, you may use the `keepPastDueSubscriptionsActive` method provided by Cashier. Typically,

this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Paddle\Cashier;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    Cashier::keepPastDueSubscriptionsActive();
}
```

{note} When a subscription is in a `past_due` state it cannot be changed until payment information has been updated. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in a `past_due` state.

Subscription Scopes

Most subscription states are also available as query scopes so that you may easily query your database for subscriptions that are in a given state:

```
// Get all active subscriptions...
$subscriptions = Subscription::query()->active()->get();

// Get all of the cancelled subscriptions for a user...
$subscriptions = $user->subscriptions()->cancelled()->get();
```

A complete list of available scopes is available below:

```
Subscription::query()->active();
Subscription::query()->onTrial();
Subscription::query()->notOnTrial();
Subscription::query()->pastDue();
Subscription::query()->recurring();
Subscription::query()->ended();
Subscription::query()->paused();
Subscription::query()->notPaused();
Subscription::query()->onPausedGracePeriod();
Subscription::query()->notOnPausedGracePeriod();
Subscription::query()->cancelled();
Subscription::query()->notCancelled();
Subscription::query()->onGracePeriod();
Subscription::query()->notOnGracePeriod();
```

Subscription Single Charges

Subscription single charges allow you to charge subscribers with a one-time charge on top of their subscriptions:

```
$response = $user->subscription('default')->charge(12.99, 'Support Add-on');
```

In contrast to single charges, this method will immediately charge the customer's stored payment method for the subscription. The charge amount should always be defined in the currency of the subscription.

Updating Payment Information

Paddle always saves a payment method per subscription. If you want to update the default payment method for a subscription, you should first generate a subscription "update URL" using the `updateUrl` method on the subscription model:

```
use App\Models\User;

$user = User::find(1);

$updateUrl = $user->subscription('default')->updateUrl();
```

Then, you may use the generated URL in combination with Cashier's provided `paddle-button` Blade component to allow the user to initiate the Paddle widget and update their payment information:

```
<x-paddle-button :url="$updateUrl" class="px-8 py-4">
    Update Card
</x-paddle-button>
```

When a user has finished updating their information, a `subscription_updated` webhook will be dispatched by Paddle and the subscription details will be updated in your application's database.

Changing Plans

After a user has subscribed to your application, they may occasionally want to change to a new subscription plan. To update the subscription plan for a user, you should pass the Paddle plan's identifier to the subscription's `swap` method:

```
use App\Models\User;

$user = User::find(1);

$user->subscription('default')->swap($premium = 34567);
```

If you would like to swap plans and immediately invoice the user instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = User::find(1);

$user->subscription('default')->swapAndInvoice($premium = 34567);
```

{note} Plans may not be swapped when a trial is active. For additional information regarding this limitation, please see the Paddle documentation.

Prorations

By default, Paddle prorates charges when swapping between plans. The `noProrate` method may be used to update the subscription's without prorating the charges:

```
$user->subscription('default')->noProrate()->swap($premium = 34567);
```

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, a project management application might charge $10 per month per project. To easily increment or decrement your subscription's quantity, use the `incrementQuantity` and `decrementQuantity` methods:

```
$user = User::find(1);

$user->subscription('default')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('default')->incrementQuantity(5);

$user->subscription('default')->decrementQuantity();

// Subtract five from the subscription's current quantity...
$user->subscription('default')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('default')->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without prorating the charges:

```
$user->subscription('default')->noProrate()->updateQuantity(10);
```

Subscription Modifiers

Subscription modifiers allow you to implement metered billing or extend subscriptions with add-ons.

For example, you might want to offer a "Premium Support" add-on with your standard subscription. You can create this modifier like so:

```
$modifier =
$user->subscription('default')->newModifier(12.99)->create();
```

The example above will add a $12.99 add-on to the subscription. By default, this charge will recur on every interval you have configured for the subscription. If you would like, you can add a readable description to the modifier using the modifier's `description` method:

```
$modifier = $user->subscription('default')->newModifier(12.99)
    ->description('Premium Support')
    ->create();
```

To illustrate how to implement metered billing using modifiers, imagine your application charges per SMS message sent by the user. First, you should create a $0 plan in your Paddle dashboard. Once the user has been subscribed to this plan, you can add modifiers representing each individual charge to the subscription:

```
$modifier = $user->subscription('default')->newModifier(0.99)
    ->description('New text message')
    ->oneTime()
    ->create();
```

As you can see, we invoked the `oneTime` method when creating this modifier. This method will ensure the modifier is only charged once and does not recur every billing interval.

Retrieving Modifiers

You may retrieve a list of all modifiers for a subscription via the `modifiers` method:

```
$modifiers = $user->subscription('default')->modifiers();

foreach ($modifiers as $modifier) {
    $modifier->amount(); // $0.99
    $modifier->description; // New text message.
}
```

Deleting Modifiers

Modifiers may be deleted by invoking the `delete` method on a `Laravel\Paddle\Modifier` instance:

```
$modifier->delete();
```

Pausing Subscriptions

To pause a subscription, call the `pause` method on the user's subscription:

```
$user->subscription('default')->pause();
```

When a subscription is paused, Cashier will automatically set the `paused_from` column in your database. This column is used to know when the `paused` method should begin

returning `true`. For example, if a customer pauses a subscription on March 1st, but the subscription was not scheduled to recur until March 5th, the `paused` method will continue to return `false` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has paused their subscription but are still on their "grace period" using the `onPausedGracePeriod` method:

```
if ($user->subscription('default')->onPausedGracePeriod()) {
    //
}
```

To resume a paused a subscription, you may call the `unpause` method on the user's subscription:

```
$user->subscription('default')->unpause();
```

{note} A subscription cannot be modified while it is paused. If you want to swap to a different plan or update quantities you must resume the subscription first.

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription('default')->cancel();
```

When a subscription is cancelled, Cashier will automatically set the `ends_at` column in your database. This column is used to know when the `subscribed` method should begin returning `false`. For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th. This is done because a user is typically allowed to continue using an application until the end of their billing cycle.

You may determine if a user has cancelled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('default')->onGracePeriod()) {
    //
}
```

If you wish to cancel a subscription immediately, you may call the `cancelNow` method on the user's subscription:

```
$user->subscription('default')->cancelNow();
```

{note} Paddle's subscriptions cannot be resumed after cancellation. If your customer wishes to resume their subscription, they will have to subscribe to a new subscription.

Subscription Trials

With Payment Method Up Front

{note} While trialing and collecting payment method details up front, Paddle prevents any subscription changes such as swapping plans or updating quantities. If you want to allow a customer to swap plans during a trial the subscription must be cancelled and recreated.

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should use the `trialDays` method when creating your subscription pay links:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $request->user()->newSubscription('default', $monthly =
12345)
                ->returnTo(route('home'))
                ->trialDays(10)
                ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

This method will set the trial period ending date on the subscription record within your application's database, as well as instruct Paddle to not begin billing the customer until after this date.

{note} If the customer's subscription is not cancelled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

You may determine if the user is within their trial period using either the `onTrial` method of the user instance or the `onTrial` method of the subscription instance. The two examples below are equivalent:

```
if ($user->onTrial('default')) {
    //
}

if ($user->subscription('default')->onTrial()) {
    //
}
```

Defining Trial Days In Paddle / Cashier

You may choose to define how many trial days your plan's receive in the Paddle dashboard or always pass them explicitly using Cashier. If you choose to define your plan's trial days in Paddle you should be aware that new subscriptions, including new subscriptions for a customer that had a subscription in the past, will always receive a trial period unless you explicitly call the `trialDays(0)` method.

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the customer record attached to your user to your desired trial ending date. This is typically done during user registration:

```
use App\Models\User;

$user = User::create([
    // ...
]);

$user->createAsCustomer([
    'trial_ends_at' => now()->addDays(10)
]);
```

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the `User` instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
use Illuminate\Http\Request;

Route::get('/user/subscribe', function (Request $request) {
    $payLink = $user->newSubscription('default', $monthly = 12345)
        ->returnTo(route('home'))
        ->create();

    return view('billing', ['payLink' => $payLink]);
});
```

To retrieve the user's trial ending date, you may use the `trialEndsAt` method. This method will return a Carbon date instance if a user is on a trial or `null` if they aren't. You may also pass an optional subscription name parameter if you would like to get the trial ending date for a specific subscription other than the default one:

```
if ($user->onTrial()) {
    $trialEndsAt = $user->trialEndsAt('main');
}
```

You may use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not created an actual subscription yet:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

{note} There is no way to extend or modify a trial period on a Paddle subscription after it has been created.

Handling Paddle Webhooks

Paddle can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is registered by the Cashier service provider. This controller will handle all incoming webhook requests.

By default, this controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Paddle subscription settings), subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any Paddle webhook event you like.

To ensure your application can handle Paddle webhooks, be sure to configure the webhook URL in the Paddle control panel. By default, Cashier's webhook controller responds to the `/paddle/webhook` URL path. The full list of all webhooks you should enable in the Paddle control panel are:

- Subscription Created
- Subscription Updated
- Subscription Cancelled
- Payment Succeeded
- Subscription Payment Succeeded

{note} Make sure you protect incoming requests with Cashier's included webhook signature verification middleware.

Webhooks & CSRF Protection

Since Paddle webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your `App\Http\Middleware\VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'paddle/*',
];
```

Webhooks & Local Development

For Paddle to be able to send your application webhooks during local development, you will need to expose your application via a site sharing service such as Ngrok or Expose. If you are developing your application locally using Laravel Sail, you may use Sail's site sharing

command.

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellation on failed charges and other common Paddle webhooks. However, if you have additional webhook events you would like to handle, you may do so by listening to the following events that are dispatched by Cashier:

- `Laravel\Paddle\Events\WebhookReceived`
- `Laravel\Paddle\Events\WebhookHandled`

Both events contain the full payload of the Paddle webhook. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you may register a listener that will handle the event:

```
<?php

namespace App\Listeners;

use Laravel\Paddle\Events\WebhookReceived;

class PaddleEventListener
{
    /**
     * Handle received Paddle webhooks.
     *
     * @param  \Laravel\Paddle\Events\WebhookReceived  $event
     * @return void
     */
    public function handle(WebhookReceived $event)
    {
        if ($event->payload['alert_name'] === 'payment_succeeded') {
            // Handle the incoming event...
        }
    }
}
```

Once your listener has been defined, you may register it within your application's `EventServiceProvider`:

```php
<?php

namespace App\Providers;

use App\Listeners\PaddleEventListener;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;
use Laravel\Paddle\Events\WebhookReceived;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        WebhookReceived::class => [
            PaddleEventListener::class,
        ],
    ];
}
```

Cashier also emit events dedicated to the type of the received webhook. In addition to the full payload from Paddle, they also contain the relevant models that were used to process the webhook such as the billable model, the subscription, or the receipt:

- `Laravel\Paddle\Events\PaymentSucceeded` - `Laravel\Paddle\Events\SubscriptionPaymentSucceeded` - `Laravel\Paddle\Events\SubscriptionCreated` - `Laravel\Paddle\Events\SubscriptionUpdated` - `Laravel\Paddle\Events\SubscriptionCancelled`

You can also override the default, built-in webhook route by defining the `CASHIER_WEBHOOK` environment variable in your application's `.env` file. This value should be the full URL to your webhook route and needs to match the URL set in your Paddle control panel:

```
CASHIER_WEBHOOK=https://example.com/my-paddle-webhook-url
```

Verifying Webhook Signatures

To secure your webhooks, you may use Paddle's webhook signatures. For convenience, Cashier automatically includes a middleware which validates that the incoming Paddle webhook request is valid.

To enable webhook verification, ensure that the `PADDLE_PUBLIC_KEY` environment variable

is defined in your application's `.env` file. The public key may be retrieved from your Paddle account dashboard.

Single Charges

Simple Charge

If you would like to make a one-time charge against a customer, you may use the `charge` method on a billable model instance to generate a pay link for the charge. The `charge` method accepts the charge amount (float) as its first argument and a charge description as its second argument:

```
use Illuminate\Http\Request;

Route::get('/store', function (Request $request) {
    return view('store', [
        'payLink' => $user->charge(12.99, 'Action Figure')
    ]);
});
```

After generating the pay link, you may use Cashier's provided `paddle-button` Blade component to allow the user to initiate the Paddle widget and complete the charge:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Buy
</x-paddle-button>
```

The `charge` method accepts an array as its third argument, allowing you to pass any options you wish to the underlying Paddle pay link creation. Please consult the Paddle documentation to learn more about the options available to you when creating charges:

```
$payLink = $user->charge(12.99, 'Action Figure', [
    'custom_option' => $value,
]);
```

Charges happen in the currency specified in the `cashier.currency` configuration option. By default, this is set to USD. You may override the default currency by defining the `CASHIER_CURRENCY` environment variable in your application's `.env` file:

```
CASHIER_CURRENCY=EUR
```

You can also override prices per currency using Paddle's dynamic pricing matching system. To do so, pass an array of prices instead of a fixed amount:

```
$payLink = $user->charge([
    'USD:19.99',
    'EUR:15.99',
], 'Action Figure');
```

Charging Products

If you would like to make a one-time charge against a specific product configured within Paddle, you may use the `chargeProduct` method on a billable model instance to generate a pay link:

```
use Illuminate\Http\Request;

Route::get('/store', function (Request $request) {
    return view('store', [
        'payLink' => $request->user()->chargeProduct($productId = 123)
    ]);
});
```

Then, you may provide the pay link to the `paddle-button` component to allow the user to initialize the Paddle widget:

```
<x-paddle-button :url="$payLink" class="px-8 py-4">
    Buy
</x-paddle-button>
```

The `chargeProduct` method accepts an array as its second argument, allowing you to pass any options you wish to the underlying Paddle pay link creation. Please consult the Paddle documentation regarding the options that are available to you when creating charges:

```
$payLink = $user->chargeProduct($productId, [
    'custom_option' => $value,
]);
```

Refunding Orders

If you need to refund a Paddle order, you may use the `refund` method. This method accepts the Paddle order ID as its first argument. You may retrieve the receipts for a given billable model using the `receipts` method:

```
use App\Models\User;

$user = User::find(1);

$receipt = $user->receipts()->first();

$refundRequestId = $user->refund($receipt->order_id);
```

You may optionally specify a specific amount to refund as well as a reason for the refund:

```
$receipt = $user->receipts()->first();

$refundRequestId = $user->refund(
    $receipt->order_id, 5.00, 'Unused product time'
);
```

{tip} You can use the `$refundRequestId` as a reference for the refund when contacting Paddle support.

Receipts

You may easily retrieve an array of a billable model's receipts via the `receipts` property:

```
use App\Models\User;

$user = User::find(1);

$receipts = $user->receipts;
```

When listing the receipts for the customer, you may use the receipt instance's methods to display the relevant receipt information. For example, you may wish to list every receipt in a table, allowing the user to easily download any of the receipts:

```
<table>
    @foreach ($receipts as $receipt)
        <tr>
            <td>{{ $receipt->paid_at->toFormattedDateString() }}</td>
            <td>{{ $receipt->amount() }}</td>
            <td><a href="{{ $receipt->receipt_url }}"
target="_blank">Download</a></td>
        </tr>
    @endforeach
</table>
```

Past & Upcoming Payments

You may use the `lastPayment` and `nextPayment` methods to retrieve and display a customer's past or upcoming payments for recurring subscriptions:

```
use App\Models\User;

$user = User::find(1);

$subscription = $user->subscription('default');

$lastPayment = $subscription->lastPayment();
$nextPayment = $subscription->nextPayment();
```

Both of these methods will return an instance of `Laravel\Paddle\Payment`; however, `nextPayment` will return `null` when the billing cycle has ended (such as when a subscription has been cancelled):

```
Next payment: {{ $nextPayment->amount() }} due on {{
$nextPayment->date()->format('d/m/Y') }}
```

Handling Failed Payments

Subscription payments fail for various reasons, such as expired cards or a card having insufficient funds. When this happens, we recommend that you let Paddle handle payment failures for you. Specifically, you may setup Paddle's automatic billing emails in your Paddle dashboard.

Alternatively, you can perform more precise customization by catching the `subscription_payment_failed` webhook and enabling the "Subscription Payment Failed" option in the Webhook settings of your Paddle dashboard:

```
<?php

namespace App\Http\Controllers;

use Laravel\Paddle\Http\Controllers\WebhookController as
CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle subscription payment failed.
     *
     * @param  array  $payload
     * @return void
     */
    public function handleSubscriptionPaymentFailed($payload)
    {
        // Handle the failed subscription payment...
    }
}
```

Testing

While testing, you should manually test your billing flow to make sure your integration works as expected.

For automated tests, including those executed within a CI environment, you may use Laravel's HTTP Client to fake HTTP calls made to Paddle. Although this does not test the actual responses from Paddle, it does provide a way to test your application without actually calling Paddle's API.

Console Tests

Introduction

In addition to simplifying HTTP testing, Laravel provides a simple API for testing your application's custom console commands.

Success / Failure Expectations

To get started, let's explore how to make assertions regarding an Artisan command's exit code. To accomplish this, we will use the `artisan` method to invoke an Artisan command from our test. Then, we will use the `assertExitCode` method to assert that the command completed with a given exit code:

```
/**
 * Test a console command.
 *
 * @return void
 */
public function test_console_command()
{
    $this->artisan('inspire')->assertExitCode(0);
}
```

You may use the `assertNotExitCode` method to assert that the command did not exit with a given exit code:

```
$this->artisan('inspire')->assertNotExitCode(1);
```

Of course, all terminal commands typically exit with a status code of `0` when they are successful and a non-zero exit code when they are not successful. Therefore, for convenience, you may utilize the `assertSuccessful` and `assertFailed` assertions to assert that a given command exited with a successful exit code or not:

```
$this->artisan('inspire')->assertSuccessful();

$this->artisan('inspire')->assertFailed();
```

Input / Output Expectations

Laravel allows you to easily "mock" user input for your console commands using the `expectsQuestion` method. In addition, you may specify the exit code and text that you expect to be output by the console command using the `assertExitCode` and `expectsOutput` methods. For example, consider the following console command:

```
Artisan::command('question', function () {
    $name = $this->ask('What is your name?');

    $language = $this->choice('Which language do you prefer?', [
        'PHP',
        'Ruby',
        'Python',
    ]);

    $this->line('Your name is '.$name.' and you prefer '.$language.'.');
});
```

You may test this command with the following test which utilizes the `expectsQuestion`, `expectsOutput`, `doesntExpectOutput`, and `assertExitCode` methods:

```
/**
 * Test a console command.
 *
 * @return void
 */
public function test_console_command()
{
    $this->artisan('question')
         ->expectsQuestion('What is your name?', 'Taylor Otwell')
         ->expectsQuestion('Which language do you prefer?', 'PHP')
         ->expectsOutput('Your name is Taylor Otwell and you prefer
PHP.')
         ->doesntExpectOutput('Your name is Taylor Otwell and you prefer
Ruby.')
         ->assertExitCode(0);
}
```

Confirmation Expectations

When writing a command which expects confirmation in the form of a "yes" or "no" answer, you may utilize the `expectsConfirmation` method:

```
$this->artisan('module:import')
    ->expectsConfirmation('Do you really wish to run this command?',
'no')
    ->assertExitCode(1);
```

Table Expectations

If your command displays a table of information using Artisan's `table` method, it can be cumbersome to write output expectations for the entire table. Instead, you may use the `expectsTable` method. This method accepts the table's headers as its first argument and the table's data as its second argument:

```
$this->artisan('users:all')
    ->expectsTable([
        'ID',
        'Email',
    ], [
        [1, 'taylor@example.com'],
        [2, 'abigail@example.com'],
    ]);
```

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
<?php

namespace App\Http\Controllers;

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

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

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

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

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

In this example, the `UserController` needs to retrieve users from a data source. So, we will **inject** a service that is able to retrieve users. In this context, our `UserRepository`

most likely uses Eloquent to retrieve user information from the database. However, since the repository is injected, we are able to easily swap it out with another implementation. We are also able to easily "mock", or create a dummy implementation of the `UserRepository` when testing our application.

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

Zero Configuration Resolution

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

```
<?php

class Service
{
    //
}

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

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

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

When To Use The Container

Thanks to zero configuration resolution, you will often type-hint dependencies on routes, controllers, event listeners, and elsewhere without ever manually interacting with the

container. For example, you might type-hint the `Illuminate\Http\Request` object on your route definition so that you can easily access the current request. Even though we never have to interact with the container to write this code, it is managing the injection of these dependencies behind the scenes:

```
use Illuminate\Http\Request;

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

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

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

Binding

Binding Basics

Simple Bindings

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

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

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

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

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

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

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

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

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

Binding A Singleton

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

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

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

Binding Scoped Singletons

The `scoped` method binds a class or interface into the container that should only be resolved one time within a given Laravel request / job lifecycle. While this method is similar to the `singleton` method, instances registered using the `scoped` method will be flushed whenever the Laravel application starts a new "lifecycle", such as when a Laravel Octane worker processes a new request or when a Laravel queue worker processes a new job:

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

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

Binding Instances

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

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

$service = new Transistor(new PodcastParser);

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

Binding Interfaces To Implementations

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

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

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

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

```
use App\Contracts\EventPusher;

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

Contextual Binding

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

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

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

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

Binding Primitives

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

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

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

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

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

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

Binding Typed Variadics

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

```php
<?php

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

class Firewall
{
    /**
     * The logger instance.
     *
     * @var \App\Services\Logger
     */
    protected $logger;

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

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

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

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

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

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

Variadic Tag Dependencies

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

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

Tagging

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

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

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

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

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

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

Extending Bindings

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

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

Resolving

The `make` Method

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

```
use App\Services\Transistor;

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

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

```
use App\Services\Transistor;

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

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

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

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

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

```
use Illuminate\Container\Container;

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

Automatic Injection

Alternatively, and importantly, you may type-hint the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, middleware, and more. Additionally, you may type-hint dependencies in the `handle` method of queued jobs. In practice, this is how most of your objects should be resolved by the container.

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

```php
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

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

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

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

Method Invocation & Injection

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

```
<?php

namespace App;

use App\Repositories\UserRepository;

class UserReport
{
    /**
     * Generate a new user report.
     *
     * @param  \App\Repositories\UserRepository  $repository
     * @return array
     */
    public function generate(UserRepository $repository)
    {
        // ...
    }
}
```

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

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

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

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

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

$result = App::call(function (UserRepository $repository) {
    // ...
});
```

Container Events

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

```
use App\Services\Transistor;

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

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

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

PSR-11

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

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

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

    //
});
```

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

Contracts

Introduction

Laravel's "contracts" are a set of interfaces that define the core services provided by the framework. For example, an `Illuminate\Contracts\Queue\Queue` contract defines the methods needed for queueing jobs, while the `Illuminate\Contracts\Mail\Mailer` contract defines the methods needed for sending e-mail.

Each contract has a corresponding implementation provided by the framework. For example, Laravel provides a queue implementation with a variety of drivers, and a mailer implementation that is powered by Symfony Mailer.

All of the Laravel contracts live in their own GitHub repository. This provides a quick reference point for all available contracts, as well as a single, decoupled package that may be utilized when building packages that interact with Laravel services.

Contracts Vs. Facades

Laravel's facades and helper functions provide a simple way of utilizing Laravel's services without needing to type-hint and resolve contracts out of the service container. In most cases, each facade has an equivalent contract.

Unlike facades, which do not require you to require them in your class' constructor, contracts allow you to define explicit dependencies for your classes. Some developers prefer to explicitly define their dependencies in this way and therefore prefer to use contracts, while other developers enjoy the convenience of facades. **In general, most applications can use facades without issue during development.**

When To Use Contracts

The decision to use contracts or facades will come down to personal taste and the tastes of your development team. Both contracts and facades can be used to create robust, well-tested Laravel applications. Contracts and facades are not mutually exclusive. Some parts of your applications may use facades while others depend on contracts. As long as you are keeping your class' responsibilities focused, you will notice very few practical differences between using contracts and facades.

In general, most applications can use facades without issue during development. If you are building a package that integrates with multiple PHP frameworks you may wish to use the `illuminate/contracts` package to define your integration with Laravel's services without the need to require Laravel's concrete implementations in your package's `composer.json` file.

How To Use Contracts

So, how do you get an implementation of a contract? It's actually quite simple.

Many types of classes in Laravel are resolved through the service container, including controllers, event listeners, middleware, queued jobs, and even route closures. So, to get an implementation of a contract, you can just "type-hint" the interface in the constructor of the class being resolved.

For example, take a look at this event listener:

```php
<?php

namespace App\Listeners;

use App\Events\OrderWasPlaced;
use App\Models\User;
use Illuminate\Contracts\Redis\Factory;

class CacheOrderInformation
{
    /**
     * The Redis factory implementation.
     *
     * @var \Illuminate\Contracts\Redis\Factory
     */
    protected $redis;

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

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderWasPlaced  $event
     * @return void
     */
    public function handle(OrderWasPlaced $event)
    {
        //
    }
}
```

When the event listener is resolved, the service container will read the type-hints on the constructor of the class, and inject the appropriate value. To learn more about registering things in the service container, check out its documentation.

Contract Reference

This table provides a quick reference to all of the Laravel contracts and their equivalent facades:

Contract	References Facade
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	`Gate`
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	`Auth`
Illuminate\Contracts\Auth\Guard	`Auth::guard()`
Illuminate\Contracts\Auth\PasswordBroker	`Password::broker()`
Illuminate\Contracts\Auth\PasswordBrokerFactory	`Password`
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	
Illuminate\Contracts\Auth\UserProvider	
Illuminate\Contracts\Bus\Dispatcher	`Bus`
Illuminate\Contracts\Bus\QueueingDispatcher	`Bus::dispatchToQueue()`
Illuminate\Contracts\Broadcasting\Factory	`Broadcast`
Illuminate\Contracts\Broadcasting\Broadcaster	`Broadcast::connection()`
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Cache\Factory	`Cache`
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	`Cache::driver()`

Contract	References Facade
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	`Config`
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	`Artisan`
Illuminate\Contracts\Container\Container	`App`
Illuminate\Contracts\Cookie\Factory	`Cookie`
Illuminate\Contracts\Cookie\QueueingFactory	`Cookie::queue()`
Illuminate\Contracts\Database\ModelIdentifier	
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	`Crypt`
Illuminate\Contracts\Events\Dispatcher	`Event`
Illuminate\Contracts\Filesystem\Cloud	`Storage::cloud()`
Illuminate\Contracts\Filesystem\Factory	`Storage`
Illuminate\Contracts\Filesystem\Filesystem	`Storage::disk()`
Illuminate\Contracts\Foundation\Application	`App`
Illuminate\Contracts\Hashing\Hasher	`Hash`
Illuminate\Contracts\Http\Kernel	
Illuminate\Contracts\Mail\MailQueue	`Mail::queue()`
Illuminate\Contracts\Mail\Mailable	
Illuminate\Contracts\Mail\Mailer	`Mail`
Illuminate\Contracts\Notifications\Dispatcher	`Notification`
Illuminate\Contracts\Notifications\Factory	`Notification`
Illuminate\Contracts\Pagination\LengthAwarePaginator	
Illuminate\Contracts\Pagination\Paginator	
Illuminate\Contracts\Pipeline\Hub	
Illuminate\Contracts\Pipeline\Pipeline	

Contract	References Facade
Illuminate\Contracts\Queue\EntityResolver	
Illuminate\Contracts\Queue\Factory	`Queue`
Illuminate\Contracts\Queue\Job	
Illuminate\Contracts\Queue\Monitor	`Queue`
Illuminate\Contracts\Queue\Queue	`Queue::connection()`
Illuminate\Contracts\Queue\QueueableCollection	
Illuminate\Contracts\Queue\QueueableEntity	
Illuminate\Contracts\Queue\ShouldQueue	
Illuminate\Contracts\Redis\Factory	`Redis`
Illuminate\Contracts\Routing\BindingRegistrar	`Route`
Illuminate\Contracts\Routing\Registrar	`Route`
Illuminate\Contracts\Routing\ResponseFactory	`Response`
Illuminate\Contracts\Routing\UrlGenerator	`URL`
Illuminate\Contracts\Routing\UrlRoutable	
Illuminate\Contracts\Session\Session	`Session::driver()`
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Htmlable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\MessageBag	
Illuminate\Contracts\Support\MessageProvider	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Support\Responsable	
Illuminate\Contracts\Translation\Loader	
Illuminate\Contracts\Translation\Translator	`Lang`
Illuminate\Contracts\Validation\Factory	`Validator`
Illuminate\Contracts\Validation\ImplicitRule	

Contract	References Facade
Illuminate\Contracts\Validation\Rule	
Illuminate\Contracts\Validation\ValidatesWhenResolved	
Illuminate\Contracts\Validation\Validator	`Validator::make()`
Illuminate\Contracts\View\Engine	
Illuminate\Contracts\View\Factory	`View`
Illuminate\Contracts\View\View	`View::make()`

Contribution Guide

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

Bug Reports

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

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

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

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

- [Laravel Application](https://github.com/laravel/laravel) - [Laravel Art](https://github.com/laravel/art) - [Laravel Documentation](https://github.com/laravel/docs) - [Laravel Dusk](https://github.com/laravel/dusk) - [Laravel Cashier Stripe](https://github.com/laravel/cashier) - [Laravel Cashier Paddle](https://github.com/laravel/cashier-paddle) - [Laravel Echo](https://github.com/laravel/echo) - [Laravel Envoy](https://github.com/laravel/envoy) - [Laravel Framework](https://github.com/laravel/framework) - [Laravel Homestead](https://github.com/laravel/homestead) - [Laravel Homestead Build Scripts](https://github.com/laravel/settler) - [Laravel Horizon](https://github.com/laravel/horizon) - [Laravel Jetstream](https://github.com/laravel/jetstream) - [Laravel Passport](https://github.com/laravel/passport) - [Laravel Sail](https://github.com/laravel/sail) - [Laravel Sanctum](https://github.com/laravel/sanctum) - [Laravel Scout](https://github.com/laravel/scout) - [Laravel Socialite](https://github.com/laravel/socialite) - [Laravel Telescope](https://github.com/laravel/telescope) - [Laravel Website](https://github.com/laravel/laravel.com-next)

Support Questions

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

- [GitHub Discussions](https://github.com/laravel/framework/discussions) - [Laracasts Forums](https://laracasts.com/discuss) - [Laravel.io Forums](https://laravel.io/forum) - [StackOverflow](https://stackoverflow.com/questions/tagged/laravel) - [Discord](https://discordapp.com/invite/mPZNm7A) - [Larachat](https://larachat.co) - [IRC](https://web.libera.chat/?nick=artisan&channels=#laravel)

Core Development Discussion

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

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

Which Branch?

All bug fixes should be sent to the latest stable branch or to the current LTS branch. Bug fixes should **never** be sent to the `master` branch unless they fix features that exist only in the upcoming release.

Minor features that are **fully backward compatible** with the current release may be sent to the latest stable branch.

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

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

Compiled Assets

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

Security Vulnerabilities

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

Coding Style

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

PHPDoc

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

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

StyleCI

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

Code of Conduct

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

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

Introduction

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

An Explanation Of The Vulnerability

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

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

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

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

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

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

Preventing CSRF Requests

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

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

```
use Illuminate\Http\Request;

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

    $token = csrf_token();

    // ...
});
```

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

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

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

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

CSRF Tokens & SPAs

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

Excluding URIs From CSRF Protection

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

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

```
<?php

namespace App\Http\Middleware;

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

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

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

X-CSRF-TOKEN

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

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

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

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

X-XSRF-TOKEN

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

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

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

Database Testing

Introduction

Laravel provides a variety of helpful tools and assertions to make it easier to test your database driven applications. In addition, Laravel model factories and seeders make it painless to create test database records using your application's Eloquent models and relationships. We'll discuss all of these powerful features in the following documentation.

Resetting The Database After Each Test

Before proceeding much further, let's discuss how to reset your database after each of your tests so that data from a previous test does not interfere with subsequent tests. Laravel's included `Illuminate\Foundation\Testing\RefreshDatabase` trait will take care of this for you. Simply use the trait on your test class:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_basic_example()
    {
        $response = $this->get('/');

        // ...
    }
}
```

Defining Model Factories

Concept Overview

First, let's talk about Eloquent model factories. When testing, you may need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Laravel allows you to define a set of default attributes for each of your Eloquent models using model factories.

To see an example of how to write a factory, take a look at the `database/factories/UserFactory.php` file in your application. This factory is included with all new Laravel applications and contains the following factory definition:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

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

As you can see, in their most basic form, factories are classes that extend Laravel's base factory class and define `definition` method. The `definition` method returns the default set of attribute values that should be applied when creating a model using the factory.

Via the `faker` property, factories have access to the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

{tip} You can set your application's Faker locale by adding a `faker_locale` option to your `config/app.php` configuration file.

Generating Factories

To create a factory, execute the `make:factory` Artisan command:

```
php artisan make:factory PostFactory
```

The new factory class will be placed in your `database/factories` directory.

Model & Factory Discovery Conventions

Once you have defined your factories, you may use the static `factory` method provided to your models by the `Illuminate\Database\Eloquent\Factories\HasFactory` trait in order to instantiate a factory instance for that model.

The `HasFactory` trait's `factory` method will use conventions to determine the proper factory for the model the trait is assigned to. Specifically, the method will look for a factory in the `Database\Factories` namespace that has a class name matching the model name and is suffixed with `Factory`. If these conventions do not apply to your particular application or factory, you may overwrite the `newFactory` method on your model to return an instance of the model's corresponding factory directly:

```
use Database\Factories\Administration\FlightFactory;

/**
 * Create a new factory instance for the model.
 *
 * @return \Illuminate\Database\Eloquent\Factories\Factory
 */
protected static function newFactory()
{
    return FlightFactory::new();
}
```

Next, define a `model` property on the corresponding factory:

```
use App\Administration\Flight;
use Illuminate\Database\Eloquent\Factories\Factory;

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

Factory States

State manipulation methods allow you to define discrete modifications that can be applied to your model factories in any combination. For example, your `Database\Factories\UserFactory` factory might contain a `suspended` state method that modifies one of its default attribute values.

State transformation methods typically call the `state` method provided by Laravel's base factory class. The `state` method accepts a closure which will receive the array of raw attributes defined for the factory and should return an array of attributes to modify:

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

Factory Callbacks

Factory callbacks are registered using the `afterMaking` and `afterCreating` methods and allow you to perform additional tasks after making or creating a model. You should register these callbacks by defining a `configure` method on your factory class. This method will be automatically called by Laravel when the factory is instantiated:

```
namespace Database\Factories;

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

class UserFactory extends Factory
{
    /**
     * Configure the model factory.
     *
     * @return $this
     */
    public function configure()
    {
        return $this->afterMaking(function (User $user) {
            //
        })->afterCreating(function (User $user) {
            //
        });
    }

    // ...
}
```

Creating Models Using Factories

Instantiating Models

Once you have defined your factories, you may use the static `factory` method provided to your models by the `Illuminate\Database\Eloquent\Factories\HasFactory` trait in order to instantiate a factory instance for that model. Let's take a look at a few examples of creating models. First, we'll use the `make` method to create models without persisting them to the database:

```
use App\Models\User;

public function test_models_can_be_instantiated()
{
    $user = User::factory()->make();

    // Use model in tests...
}
```

You may create a collection of many models using the `count` method:

```
$users = User::factory()->count(3)->make();
```

Applying States

You may also apply any of your states to the models. If you would like to apply multiple state transformations to the models, you may simply call the state transformation methods directly:

```
$users = User::factory()->count(5)->suspended()->make();
```

Overriding Attributes

If you would like to override some of the default values of your models, you may pass an

array of values to the `make` method. Only the specified attributes will be replaced while the rest of the attributes remain set to their default values as specified by the factory:

```
$user = User::factory()->make([
    'name' => 'Abigail Otwell',
]);
```

Alternatively, the `state` method may be called directly on the factory instance to perform an inline state transformation:

```
$user = User::factory()->state([
    'name' => 'Abigail Otwell',
])->make();
```

{tip} Mass assignment protection is automatically disabled when creating models using factories.

Persisting Models

The `create` method instantiates model instances and persists them to the database using Eloquent's `save` method:

```
use App\Models\User;

public function test_models_can_be_persisted()
{
    // Create a single App\Models\User instance...
    $user = User::factory()->create();

    // Create three App\Models\User instances...
    $users = User::factory()->count(3)->create();

    // Use model in tests...
}
```

You may override the factory's default model attributes by passing an array of attributes to the `create` method:

```
$user = User::factory()->create([
    'name' => 'Abigail',
]);
```

Sequences

Sometimes you may wish to alternate the value of a given model attribute for each created model. You may accomplish this by defining a state transformation as a sequence. For example, you may wish to alternate the value of an `admin` column between `Y` and `N` for each created user:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
                ->count(10)
                ->state(new Sequence(
                    ['admin' => 'Y'],
                    ['admin' => 'N'],
                ))
                ->create();
```

In this example, five users will be created with an `admin` value of `Y` and five users will be created with an `admin` value of `N`.

If necessary, you may include a closure as a sequence value. The closure will be invoked each time the sequence needs a new value:

```
$users = User::factory()
                ->count(10)
                ->state(new Sequence(
                    fn ($sequence) => ['role' =>
UserRoles::all()->random()],
                ))
                ->create();
```

Within a sequence closure, you may access the `$index` or `$count` properties on the sequence instance that is injected into the closure. The `$index` property contains the

number of iterations through the sequence that have occurred thus far, while the `$count` property contains the total number of times the sequence will be invoked:

```
$users = User::factory()
                ->count(10)
                ->sequence(fn ($sequence) => ['name' => 'Name
'.$sequence->index])
                ->create();
```

Factory Relationships

Has Many Relationships

Next, let's explore building Eloquent model relationships using Laravel's fluent factory methods. First, let's assume our application has an `App\Models\User` model and an `App\Models\Post` model. Also, let's assume that the `User` model defines a `hasMany` relationship with `Post`. We can create a user that has three posts using the `has` method provided by the Laravel's factories. The `has` method accepts a factory instance:

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

$user = User::factory()
            ->has(Post::factory()->count(3))
            ->create();
```

By convention, when passing a `Post` model to the `has` method, Laravel will assume that the `User` model must have a `posts` method that defines the relationship. If necessary, you may explicitly specify the name of the relationship that you would like to manipulate:

```
$user = User::factory()
            ->has(Post::factory()->count(3), 'posts')
            ->create();
```

Of course, you may perform state manipulations on the related models. In addition, you may pass a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
            ->has(
                Post::factory()
                        ->count(3)
                        ->state(function (array $attributes, User $user)
{
                            return ['user_type' => $user->type];
                        })
            )
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to build relationships. For example, the following example will use convention to determine that the related models should be created via a `posts` relationship method on the `User` model:

```
$user = User::factory()
            ->hasPosts(3)
            ->create();
```

When using magic methods to create factory relationships, you may pass an array of attributes to override on the related models:

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

You may provide a closure based state transformation if your state change requires access to the parent model:

```
$user = User::factory()
            ->hasPosts(3, function (array $attributes, User $user) {
                return ['user_type' => $user->type];
            })
            ->create();
```

Belongs To Relationships

Now that we have explored how to build "has many" relationships using factories, let's explore the inverse of the relationship. The `for` method may be used to define the parent model that factory created models belong to. For example, we can create three `App\Models\Post` model instances that belong to a single user:

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

$posts = Post::factory()
            ->count(3)
            ->for(User::factory()->state([
                'name' => 'Jessica Archer',
            ]))
            ->create();
```

If you already have a parent model instance that should be associated with the models you are creating, you may pass the model instance to the `for` method:

```
$user = User::factory()->create();

$posts = Post::factory()
            ->count(3)
            ->for($user)
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define "belongs to" relationships. For example, the following example will use convention to

determine that the three posts should belong to the `user` relationship on the `Post` model:

```
$posts = Post::factory()
            ->count(3)
            ->forUser([
                'name' => 'Jessica Archer',
            ])
            ->create();
```

Many To Many Relationships

Like has many relationships, "many to many" relationships may be created using the `has` method:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
            ->has(Role::factory()->count(3))
            ->create();
```

Pivot Table Attributes

If you need to define attributes that should be set on the pivot / intermediate table linking the models, you may use the `hasAttached` method. This method accepts an array of pivot table attribute names and values as its second argument:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
            ->hasAttached(
                Role::factory()->count(3),
                ['active' => true]
            )
            ->create();
```

You may provide a closure based state transformation if your state change requires access to the related model:

```
$user = User::factory()
            ->hasAttached(
                Role::factory()
                    ->count(3)
                    ->state(function (array $attributes, User $user) {
                        return ['name' => $user->name.' Role'];
                    }),
                ['active' => true]
            )
            ->create();
```

If you already have model instances that you would like to be attached to the models you are creating, you may pass the model instances to the `hasAttached` method. In this example, the same three roles will be attached to all three users:

```
$roles = Role::factory()->count(3)->create();

$user = User::factory()
            ->count(3)
            ->hasAttached($roles, ['active' => true])
            ->create();
```

Using Magic Methods

For convenience, you may use Laravel's magic factory relationship methods to define many to many relationships. For example, the following example will use convention to determine that the related models should be created via a `roles` relationship method on the `User` model:

```
$user = User::factory()
            ->hasRoles(1, [
                'name' => 'Editor'
            ])
            ->create();
```

Polymorphic Relationships

Polymorphic relationships may also be created using factories. Polymorphic "morph many" relationships are created in the same way as typical "has many" relationships. For example, if a `App\Models\Post` model has a `morphMany` relationship with a `App\Models\Comment` model:

```
use App\Models\Post;

$post = Post::factory()->hasComments(3)->create();
```

Morph To Relationships

Magic methods may not be used to create `morphTo` relationships. Instead, the `for` method must be used directly and the name of the relationship must be explicitly provided. For example, imagine that the `Comment` model has a `commentable` method that defines a `morphTo` relationship. In this situation, we may create three comments that belong to a single post by using the `for` method directly:

```
$comments = Comment::factory()->count(3)->for(
    Post::factory(), 'commentable'
)->create();
```

Polymorphic Many To Many Relationships

Polymorphic "many to many" (`morphToMany` / `morphedByMany`) relationships may be created just like non-polymorphic "many to many" relationships:

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
            ->hasAttached(
                Tag::factory()->count(3),
                ['public' => true]
            )
            ->create();
```

Of course, the magic `has` method may also be used to create polymorphic "many to many"

relationships:

```
$videos = Video::factory()
            ->hasTags(3, ['public' => true])
            ->create();
```

Defining Relationships Within Factories

To define a relationship within your model factory, you will typically assign a new factory instance to the foreign key of the relationship. This is normally done for the "inverse" relationships such as `belongsTo` and `morphTo` relationships. For example, if you would like to create a new user when creating a post, you may do the following:

```
use App\Models\User;

/**
 * Define the model's default state.
 *
 * @return array
 */
public function definition()
{
    return [
        'user_id' => User::factory(),
        'title' => $this->faker->title(),
        'content' => $this->faker->paragraph(),
    ];
}
```

If the relationship's columns depend on the factory that defines it you may assign a closure to an attribute. The closure will receive the factory's evaluated attribute array:

```
/**
 * Define the model's default state.
 *
 * @return array
 */
public function definition()
{
    return [
        'user_id' => User::factory(),
        'user_type' => function (array $attributes) {
            return User::find($attributes['user_id'])->type;
        },
        'title' => $this->faker->title(),
        'content' => $this->faker->paragraph(),
    ];
}
```

Running Seeders

If you would like to use database seeders to populate your database during a feature test, you may invoke the `seed` method. By default, the `seed` method will execute the `DatabaseSeeder`, which should execute all of your other seeders. Alternatively, you pass a specific seeder class name to the `seed` method:

```php
<?php

namespace Tests\Feature;

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Test creating a new order.
     *
     * @return void
     */
    public function test_orders_can_be_created()
    {
        // Run the DatabaseSeeder...
        $this->seed();

        // Run a specific seeder...
        $this->seed(OrderStatusSeeder::class);

        // ...

        // Run an array of specific seeders...
        $this->seed([
            OrderStatusSeeder::class,
            TransactionStatusSeeder::class,
            // ...
        ]);
    }
}
```

Alternatively, you may instruct Laravel to automatically seed the database before each test that uses the `RefreshDatabase` trait. You may accomplish this by defining a `$seed` property on your base test class:

```php
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    /**
     * Indicates whether the default seeder should run before each test.
     *
     * @var bool
     */
    protected $seed = true;
}
```

When the `$seed` property is `true`, the test will run the `Database\Seeders\DatabaseSeeder` class before each test that uses the `RefreshDatabase` trait. However, you may specify a specific seeder that should be executed by defining a `$seeder` property on your test class:

```php
use Database\Seeders\OrderStatusSeeder;

/**
 * Run a specific seeder before each test.
 *
 * @var string
 */
protected $seeder = OrderStatusSeeder::class;
```

Available Assertions

Laravel provides several database assertions for your PHPUnit feature tests. We'll discuss each of these assertions below.

assertDatabaseCount

Assert that a table in the database contains the given number of records:

```
$this->assertDatabaseCount('users', 5);
```

assertDatabaseHas

Assert that a table in the database contains records matching the given key / value query constraints:

```
$this->assertDatabaseHas('users', [
    'email' => 'sally@example.com',
]);
```

assertDatabaseMissing

Assert that a table in the database does not contain records matching the given key / value query constraints:

```
$this->assertDatabaseMissing('users', [
    'email' => 'sally@example.com',
]);
```

assertDeleted

The `assertDeleted` asserts that a given Eloquent model has been deleted from the database:

```
use App\Models\User;

$user = User::find(1);

$user->delete();

$this->assertDeleted($user);
```

The `assertSoftDeleted` method may be used to assert a given Eloquent model has been "soft deleted":

```
$this->assertSoftDeleted($user);
```

assertModelExists

Assert that a given model exists in the database:

```
use App\Models\User;

$user = User::factory()->create();

$this->assertModelExists($user);
```

assertModelMissing

Assert that a given model does not exist in the database:

```
use App\Models\User;

$user = User::factory()->create();

$user->delete();

$this->assertModelMissing($user);
```

Deployment

Introduction

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

Server Requirements

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

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

Server Configuration

Nginx

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

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

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

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-Content-Type-Options "nosniff";

    index index.php;

    charset utf-8;

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

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

    error_page 404 /index.php;

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

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

Optimization

Autoloader Optimization

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

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

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

Optimizing Configuration Loading

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

```
php artisan config:cache
```

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

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

Optimizing Route Loading

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

```
php artisan route:cache
```

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

Optimizing View Loading

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

```
php artisan view:cache
```

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

Debug Mode

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

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

Deploying With Forge / Vapor

Laravel Forge

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

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

Laravel Vapor

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

Prologue

- Release Notes
- Upgrade Guide
- Contribution Guide

Getting Started

Architecture Concepts

The Basics

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

Digging Deeper

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

. Security

- Authentication
- Authorization
- Email Verification
- Encryption
- Hashing
- Password Reset

Database

- Getting Started
- Query Builder
- Pagination
- Migrations
- Seeding
- Redis

Eloquent ORM

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

Testing

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

Installation

To get started, you should install Google Chrome and add the `laravel/dusk` Composer dependency to your project:

```
composer require --dev laravel/dusk
```

{note} If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.

After installing the Dusk package, execute the `dusk:install` Artisan command. The `dusk:install` command will create a `tests/Browser` directory and an example Dusk test:

```
php artisan dusk:install
```

Next, set the `APP_URL` environment variable in your application's `.env` file. This value should match the URL you use to access your application in a browser.

{tip} If you are using Laravel Sail to manage your local development environment, please also consult the Sail documentation on configuring and running Dusk tests.

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is included with Laravel Dusk, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 86

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all

# Install the version of ChromeDriver that matches the detected version
of Chrome / Chromium for your OS...
php artisan dusk:chrome-driver --detect
```

{note} Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone ChromeDriver installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the ChromeDriver:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In

addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Database Migrations

Most of the tests you write will interact with pages that retrieve data from your application's database; however, your Dusk tests should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which will not be applicable or available across HTTP requests. Instead, use the `DatabaseMigrations` trait, which re-migrates the database for each test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
}
```

{note} SQLite in-memory databases may not be used when executing Dusk tests. Since the browser executes within its own process, it will not be able to access the in-memory databases of other processes.

Running Tests

To run your browser tests, execute the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, such as allowing you to only run the tests for a given group:

```
php artisan dusk --group=foo
```

{tip} If you are using Laravel Sail to manage your local development environment, please consult the Sail documentation on configuring and running Dusk tests.

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class to reflect the correct port:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a `.env.dusk.local` file.

When running tests, Dusk will back-up your `.env` file and rename your Dusk environment to `.env`. Once the tests have completed, your `.env` file will be restored.

Browser Basics

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, you may call the `browse` method from within your Dusk test:

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

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

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                    ->type('email', $user->email)
                    ->type('password', 'password')
                    ->press('Login')
                    ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a closure. A browser instance will automatically be passed to this closure by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, simply add more browser arguments to the

signature of the closure given to the `browse` method:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home')
          ->waitForText('Message');

    $second->loginAs(User::find(2))
           ->visit('/home')
           ->waitForText('Message')
           ->type('message', 'Hey Taylor')
           ->press('Send');

    $first->waitForText('Hey Taylor')
          ->assertSee('Jeffrey Way');
});
```

Navigation

The `visit` method may be used to navigate to a given URI within your application:

```
$browser->visit('/login');
```

You may use the `visitRoute` method to navigate to a named route:

```
$browser->visitRoute('login');
```

You may navigate "back" and "forward" using the `back` and `forward` methods:

```
$browser->back();

$browser->forward();
```

You may use the `refresh` method to refresh the page:

```
$browser->refresh();
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

The `fitContent` method will resize the browser window to match the size of its content:

```
$browser->fitContent();
```

When a test fails, Dusk will automatically resize the browser to fit the content prior to taking a screenshot. You may disable this feature by calling the `disableFitOnFailure` method within your test:

```
$browser->disableFitOnFailure();
```

You may use the `move` method to move the browser window to a different position on your screen:

```
$browser->move($x = 100, $y = 100);
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register Dusk's browser macros.
     *
     * @return void
     */
    public function boot()
    {
        Browser::macro('scrollToElement', function ($element = null) {
            $this->script("$('html, body').animate({ scrollTop:
$('$element').offset().top }, 0);");

            return $this;
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a closure as its second. The macro's closure will be executed when calling the macro as a method on a `Browser` instance:

```
$this->browse(function ($browser) use ($user) {
    $browser->visit('/pay')
            ->scrollToElement('#credit-card-details')
            ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with your application's login screen during every test. The `loginAs` method accepts a primary key associated with your authenticatable model or an authenticatable model instance:

```
use App\Models\User;

$this->browse(function ($browser) {
    $browser->loginAs(User::find(1))
          ->visit('/home');
});
```

{note} After using the `loginAs` method, the user session will be maintained for all tests within the file.

Cookies

You may use the `cookie` method to get or set an encrypted cookie's value. By default, all of the cookies created by Laravel are encrypted:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

You may use the `plainCookie` method to get or set an unencrypted cookie's value:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

You may use the `deleteCookie` method to delete the given cookie:

```
$browser->deleteCookie('name');
```

Executing JavaScript

You may use the `script` method to execute arbitrary JavaScript statements within the browser:

```
$browser->script('document.documentElement.scrollTop = 0');

$browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);

$output = $browser->script('return window.location.pathname');
```

Taking A Screenshot

You may use the `screenshot` method to take a screenshot and store it with the given filename. All screenshots will be stored within the `tests/Browser/screenshots` directory:

```
$browser->screenshot('filename');
```

Storing Console Output To Disk

You may use the `storeConsoleLog` method to write the current browser's console output to disk with the given filename. Console output will be stored within the `tests/Browser/console` directory:

```
$browser->storeConsoleLog('filename');
```

Storing Page Source To Disk

You may use the `storeSource` method to write the current page's source to disk with the given filename. The page source will be stored within the `tests/Browser/source` directory:

```
$browser->storeSource('filename');
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, when interacting with a Dusk browser, prefix the selector with `@` to manipulate the attached element within your test:

```
// HTML...

<button dusk="login-button">Login</button>

// Test...

$browser->click('@login-button');
```

Text, Values, & Attributes

Retrieving & Setting Values

Dusk provides several methods for interacting with the current value, display text, and

attributes of elements on the page. For example, to get the "value" of an element that matches a given CSS or Dusk selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

You may use the `inputValue` method to get the "value" of an input element that has a given field name:

```
$value = $browser->inputValue('field');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve the value of an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Interacting With Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an `input` or `textarea` field with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
        ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

You can instruct Dusk to type slowly using the `typeSlowly` method. By default, Dusk will pause for 100 milliseconds between key presses. To customize the amount of time between key presses, you may pass the appropriate number of milliseconds as the third argument to the method:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');

$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

You may use the `appendSlowly` method to append text slowly:

```
$browser->type('tags', 'foo')
        ->appendSlowly('tags', ', bar, baz');
```

Dropdowns

To select a value available on a `select` element, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second argument:

```
$browser->select('size');
```

By providing an array as the second argument to the `select` method, you can instruct the method to select multiple options:

```
$browser->select('categories', ['Art', 'Music']);
```

Checkboxes

To "check" a checkbox input, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');
```

The `uncheck` method may be used to "uncheck" a checkbox input:

```
$browser->uncheck('terms');
```

Radio Buttons

To "select" a `radio` input option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `radio` input with matching `name` and `value` attributes:

```
$browser->radio('size', 'large');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If a CSS selector match can't be found, Dusk will search for a `file` input with a matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```

{note} The attach function requires the `Zip` PHP extension to be installed and enabled on your server.

Pressing Buttons

The `press` method may be used to click a button element on the page. The first argument given to the `press` method may be either the display text of the button or a CSS / Dusk selector:

```
$browser->press('Login');
```

When submitting forms, many application's disable the form's submission button after it is pressed and then re-enable the button when the form submission's HTTP request is complete. To press a button and wait for the button to be re-enabled, you may use the `pressAndWaitFor` method:

```
// Press the button and wait a maximum of 5 seconds for it to be
enabled...
$browser->pressAndWaitFor('Save');

// Press the button and wait a maximum of 1 second for it to be
enabled...
$browser->pressAndWaitFor('Save', 1);
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

You may use the `seeLink` method to determine if a link with the given display text is visible on the page:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```

{note} These methods interact with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Using The Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may instruct Dusk to hold modifier keys while entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `swift` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Another valuable use case for the `keys` method is sending a "keyboard shortcut" combination to the primary CSS selector for your application:

```
$browser->keys('.app', ['{command}', 'j']);
```

{tip} All modifier keys such as `{command}` are wrapped in `{}` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be found on GitHub.

Using The Mouse

Clicking On Elements

The `click` method may be used to click on an element matching the given CSS or Dusk selector:

```
$browser->click('.selector');
```

The `clickAtXPath` method may be used to click on an element matching the given XPath expression:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

The `clickAtPoint` method may be used to click on the topmost element at a given pair of coordinates relative to the viewable area of the browser:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

The `doubleClick` method may be used to simulate the double click of a mouse:

```
$browser->doubleClick();
```

The `rightClick` method may be used to simulate the right click of a mouse:

```
$browser->rightClick();

$browser->rightClick('.selector');
```

The `clickAndHold` method may be used to simulate a mouse button being clicked and held down. A subsequent call to the `releaseMouse` method will undo this behavior and release the mouse button:

```
$browser->clickAndHold()
        ->pause(1000)
        ->releaseMouse();
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given CSS or Dusk selector:

```
$browser->mouseover('.selector');
```

Drag & Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', $pixels = 10);
$browser->dragRight('.selector', $pixels = 10);
$browser->dragUp('.selector', $pixels = 10);
$browser->dragDown('.selector', $pixels = 10);
```

Finally, you may drag an element by a given offset:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs. For example, you may use the `waitForDialog` method to wait for a JavaScript dialog to appear. This method accepts an optional argument indicating how many seconds to wait for the dialog to appear:

```
$browser->waitForDialog($seconds = null);
```

The `assertDialogOpened` method may be used to assert that a dialog has been displayed and contains the given message:

```
$browser->assertDialogOpened('Dialog message');
```

If the JavaScript dialog contains a prompt, you may use the `typeInDialog` method to type a value into the prompt:

```
$browser->typeInDialog('Hello World');
```

To close an open JavaScript dialog by clicking the "OK" button, you may invoke the `acceptDialog` method:

```
$browser->acceptDialog();
```

To close an open JavaScript dialog by clicking the "Cancel" button, you may invoke the `dismissDialog` method:

```
$browser->dismissDialog();
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to

accomplish this. All operations performed within the closure given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

You may occasionally need to execute assertions outside of the current scope. You may use the `elsewhere` and `elsewhereWhenAvailable` methods to accomplish this:

```
 $browser->with('.table', function ($table) {
    // Current scope is `body .table`...

    $browser->elsewhere('.page-title', function ($title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });

    $browser->elsewhereWhenAvailable('.page-title', function ($title) {
        // Current scope is `body .page-title`...
        $title->assertSee('Hello World');
    });
 });
```

Waiting For Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to become visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you just need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

Waiting For Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS or Dusk selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

You may also wait until the element matching the given selector contains the given text:

```
// Wait a maximum of five seconds for the selector to contain the given
text...
$browser->waitForTextIn('.selector', 'Hello World');

// Wait a maximum of one second for the selector to contain the given
text...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

You may also wait until the element matching the given selector is missing from the page:

```
// Wait a maximum of five seconds until the selector is missing...
$browser->waitUntilMissing('.selector');

// Wait a maximum of one second until the selector is missing...
$browser->waitUntilMissing('.selector', 1);
```

Or, you may wait until the element matching the given selector is enabled or disabled:

```
// Wait a maximum of five seconds until the selector is enabled...
$browser->waitUntilEnabled('.selector');

// Wait a maximum of one second until the selector is enabled...
$browser->waitUntilEnabled('.selector', 1);

// Wait a maximum of five seconds until the selector is disabled...
$browser->waitUntilDisabled('.selector');

// Wait a maximum of one second until the selector is disabled...
$browser->waitUntilDisabled('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for an element to appear that matches a given selector and then interact with the element. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used to accomplish this. All element operations performed within the given closure will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
          ->press('OK');
});
```

Waiting For Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

You may use the `waitUntilMissingText` method to wait until the displayed text has been removed from the page:

```
// Wait a maximum of five seconds for the text to be removed...
$browser->waitUntilMissingText('Hello World');

// Wait a maximum of one second for the text to be removed...
$browser->waitUntilMissingText('Hello World', 1);
```

Waiting For Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting On The Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

The `waitForLocation` method can also be used to wait for the current window location to be a fully qualified URL:

```
$browser->waitForLocation('https://example.com/path');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to make assertions after a page has been reloaded, use the `waitForReload` method:

```
$browser->click('.some-action')
        ->waitForReload()
        ->assertSee('something');
```

Waiting On JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitUntil` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

Waiting On Vue Expressions

The `waitUntilVue` and `waitUntilVueIsNot` methods may be used to wait until a Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitUntilVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitUntilVueIsNot('user.name', null, '@user');
```

Waiting With A Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given closure to return `true`. The `waitUsing` method accepts the maximum number of seconds to wait, the interval at which the closure should

be evaluated, the closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Scrolling An Element Into View

Sometimes you may not be able to click on an element because it is outside of the viewable area of the browser. The `scrollIntoView` method will scroll the browser window until the element at the given selector is within the view:

```
$browser->scrollIntoView('.selector')
        ->click('.selector');
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

[assertTitle](#assert-title) [assertTitleContains](#assert-title-contains) [assertUrlIs](#assert-url-is) [assertSchemeIs](#assert-scheme-is) [assertSchemeIsNot](#assert-scheme-is-not) [assertHostIs](#assert-host-is) [assertHostIsNot](#assert-host-is-not) [assertPortIs](#assert-port-is) [assertPortIsNot](#assert-port-is-not) [assertPathBeginsWith](#assert-path-begins-with) [assertPathIs](#assert-path-is) [assertPathIsNot](#assert-path-is-not) [assertRouteIs](#assert-route-is) [assertQueryStringHas](#assert-query-string-has) [assertQueryStringMissing](#assert-query-string-missing) [assertFragmentIs](#assert-fragment-is) [assertFragmentBeginsWith](#assert-fragment-begins-with) [assertFragmentIsNot](#assert-fragment-is-not) [assertHasCookie](#assert-has-cookie) [assertHasPlainCookie](#assert-has-plain-cookie) [assertCookieMissing](#assert-cookie-missing) [assertPlainCookieMissing](#assert-plain-cookie-missing) [assertCookieValue](#assert-cookie-value) [assertPlainCookieValue](#assert-plain-cookie-value) [assertSee](#assert-see) [assertDontSee](#assert-dont-see) [assertSeeIn](#assert-see-in) [assertDontSeeIn](#assert-dont-see-in) [assertSeeAnythingIn](#assert-see-anything-in) [assertSeeNothingIn](#assert-see-nothing-in) [assertScript](#assert-script) [assertSourceHas](#assert-source-has) [assertSourceMissing](#assert-source-missing) [assertSeeLink](#assert-see-link) [assertDontSeeLink](#assert-dont-see-link) [assertInputValue](#assert-input-value) [assertInputValueIsNot](#assert-input-value-is-not) [assertChecked](#assert-checked) [assertNotChecked](#assert-not-checked) [assertRadioSelected](#assert-radio-selected) [assertRadioNotSelected](#assert-radio-not-selected) [assertSelected](#assert-selected) [assertNotSelected](#assert-not-selected) [assertSelectHasOptions](#assert-select-has-options) [assertSelectMissingOptions](#assert-select-missing-options) [assertSelectHasOption](#assert-select-has-option) [assertSelectMissingOption](#assert-select-missing-option) [assertValue](#assert-value) [assertValueIsNot](#assert-value-is-not) [assertAttribute](#assert-attribute) [assertAttributeContains](#assert-attribute-contains) [assertAriaAttribute](#assert-aria-attribute) [assertDataAttribute](#assert-data-attribute) [assertVisible](#assert-visible) [assertPresent](#assert-present) [assertNotPresent](#assert-not-present) [assertMissing](#assert-missing) [assertInputPresent](#assert-input-present) [assertInputMissing](#assert-input-missing) [assertDialogOpened](#assert-dialog-opened) [assertEnabled](#assert-enabled) [assertDisabled](#assert-disabled) [assertButtonEnabled](#assert-button-enabled) [assertButtonDisabled](#assert-button-disabled) [assertFocused](#assert-focused) [assertNotFocused](#assert-not-focused) [assertAuthenticated](#assert-authenticated) [assertGuest](#assert-guest) [assertAuthenticatedAs](#assert-authenticated-as) [assertVue](#assert-vue) [assertVueIsNot](#assert-vue-is-not) [assertVueContains](#assert-vue-contains) [assertVueDoesNotContain](#assert-vue-does-not-contain)

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith('/home');
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the URL's current hash fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the URL's current hash fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the URL's current hash fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given encrypted cookie is present:

```
$browser->assertHasCookie($name);
```

assertHasPlainCookie

Assert that the given unencrypted cookie is present:

```
$browser->assertHasPlainCookie($name);
```

assertCookieMissing

Assert that the given encrypted cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertPlainCookieMissing

Assert that the given unencrypted cookie is not present:

```
$browser->assertPlainCookieMissing($name);
```

assertCookieValue

Assert that an encrypted cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSeeAnythingIn

Assert that any text is present within the selector:

```
$browser->assertSeeAnythingIn($selector);
```

assertSeeNothingIn

Assert that no text is present within the selector:

```
$browser->assertSeeNothingIn($selector);
```

assertScript

Assert that the given JavaScript expression evaluates to the given value:

```
$browser->assertScript('window.isLoaded')
        ->assertScript('document.readyState', 'complete');
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValueIsNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertSelectMissingOption

Assert that the given value is not available to be selected:

```
$browser->assertSelectMissingOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertValueIsNot

Assert that the element matching the given selector does not have the given value:

```
$browser->assertValueIsNot($selector, $value);
```

assertAttribute

Assert that the element matching the given selector has the given value in the provided attribute:

```
$browser->assertAttribute($selector, $attribute, $value);
```

assertAttributeContains

Assert that the element matching the given selector contains the given value in the provided attribute:

```
$browser->assertAttributeContains($selector, $attribute, $value);
```

assertAriaAttribute

Assert that the element matching the given selector has the given value in the provided aria attribute:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

For example, given the markup `<button aria-label="Add"></button>`, you may assert against the `aria-label` attribute like so:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

assertDataAttribute

Assert that the element matching the given selector has the given value in the provided data attribute:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

For example, given the markup `<tr id="row-1" data-content="attendees"></tr>`, you may assert against the `data-label` attribute like so:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present in the source:

```
$browser->assertPresent($selector);
```

assertNotPresent

Assert that the element matching the given selector is not present in the source:

```
$browser->assertNotPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertInputPresent

Assert that an input with the given name is present:

```
$browser->assertInputPresent($name);
```

assertInputMissing

Assert that an input with the given name is not present in the source:

```
$browser->assertInputMissing($name);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertAuthenticated

Assert that the user is authenticated:

```
$browser->assertAuthenticated();
```

assertGuest

Assert that the user is not authenticated:

```
$browser->assertGuest();
```

assertAuthenticatedAs

Assert that the user is authenticated as the given user:

```
$browser->assertAuthenticatedAs($user);
```

assertVue

Dusk even allows you to make assertions on the state of Vue component data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/**
 * A basic Vue test example.
 *
 * @return void
 */
public function testVue()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
                ->assertVue('user.name', 'Taylor', '@profile-
component');
    });
}
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector =
null);
```

assertVueDoesNotContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector
= null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Dusk Pages allow you to define expressive actions that may then be performed on a given page via a single method. Pages also allow you to define short-cuts to common selectors for your application or for a single page.

Generating Pages

To generate a page object, execute the `dusk:page` Artisan command. All page objects will be placed in your application's `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be discussed in more detail below.

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**
 * Get the URL for the page.
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. It is not actually necessary to place anything within this method; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

Navigating To Pages

Once a page has been defined, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being

redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
        ->clickLink('Create Playlist')
        ->on(new CreatePlaylist)
        ->assertSee('@create');
```

Shorthand Selectors

The `elements` method within page classes 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]',
    ];
}
```

Once the shortcut has been defined, you may use the shorthand selector anywhere you would typically use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global

shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  string  $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
                ->check('share')
                ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed as the first argument to custom page methods:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, execute the `dusk:component` Artisan command. New components are placed in the `tests/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }
```

```
    /**
     * Assert that the browser page contains the component.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  int  $year
     * @param  int  $month
     * @param  int  $day
     * @return void
     */
    public function selectDate(Browser $browser, $year, $month, $day)
    {
        $browser->click('@date-field')
                ->within('@year-list', function ($browser) use ($year) {
                    $browser->click($year);
                })
                ->within('@month-list', function ($browser) use ($month)
{
                    $browser->click($month);
                })
                ->within('@day-list', function ($browser) use ($day) {
```

```
                    $browser->click($day);
                });
    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```
<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->within(new DatePicker, function ($browser) {
                        $browser->selectDate(2019, 1, 30);
                    })
                    ->assertSee('January');
        });
    }
}
```

Continuous Integration

{note} Most Dusk continuous integration configurations expect your Laravel application to be served using the built-in PHP development server on port 8000. Therefore, before continuing, you should ensure that your continuous integration environment has an `APP_URL` environment variable value of `http://127.0.0.1:8000`.

Heroku CI

To run Dusk tests on Heroku CI, add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url":
"https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-
linux > /dev/null 2>&1 &' && nohup bash -c 'php artisan serve --no-
reload > /dev/null 2>&1 &' && php artisan dusk"
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on Travis CI, use the following `.travis.yml` configuration. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-
port=9222 http://localhost &
  - php artisan serve --no-reload &

script:
  - php artisan dusk
```

GitHub Actions

If you are using Github Actions to run your Dusk tests, you may use the following configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
name: CI
on: [push]
jobs:

  dusk-php:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: |
          sudo systemctl start mysql
```

```
          mysql --user="root" --password="root" -e "CREATE DATABASE 'my-
database' character set UTF8mb4 collate utf8mb4_bin;"
      - name: Install Composer Dependencies
        run: composer install --no-progress --prefer-dist --optimize-
autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver `/opt/google/chrome/chrome -
-version | cut -d " " -f3 | cut -d "." -f1`
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux &
      - name: Run Laravel Server
        run: php artisan serve --no-reload &
      - name: Run Dusk Tests
        env:
          APP_URL: "http://127.0.0.1:8000"
        run: php artisan dusk
      - name: Upload Screenshots
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: screenshots
          path: tests/Browser/screenshots
      - name: Upload Console Logs
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: console
          path: tests/Browser/console
```

Eloquent: API Resources

- Resource Responses

Introduction

When building an API, you may need a transformation layer that sits between your Eloquent models and the JSON responses that are actually returned to your application's users. For example, you may wish to display certain attributes for a subset of users and not others, or you may wish to always include certain relationships in the JSON representation of your models. Eloquent's resource classes allow you to expressively and easily transform your models and model collections into JSON.

Of course, you may always convert Eloquent models or collections to JSON using their `toJson` methods; however, Eloquent resources provide more granular and robust control over the JSON serialization of your models and their relationships.

Generating Resources

To generate a resource class, you may use the `make:resource` Artisan command. By default, resources will be placed in the `app/Http/Resources` directory of your application. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource UserResource
```

Resource Collections

In addition to generating resources that transform individual models, you may generate resources that are responsible for transforming collections of models. This allows your JSON responses to include links and other meta information that is relevant to an entire collection of a given resource.

To create a resource collection, you should use the `--collection` flag when creating the resource. Or, including the word `Collection` in the resource name will indicate to Laravel that it should create a collection resource. Collection resources extend the `Illuminate\Http\Resources\Json\ResourceCollection` class:

```
php artisan make:resource User --collection

php artisan make:resource UserCollection
```

Concept Overview

{tip} This is a high-level overview of resources and resource collections. You are highly encouraged to read the other sections of this documentation to gain a deeper understanding of the customization and power offered to you by resources.

Before diving into all of the options available to you when writing resources, let's first take a high-level look at how resources are used within Laravel. A resource class represents a single model that needs to be transformed into a JSON structure. For example, here is a simple `UserResource` resource class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Every resource class defines a `toArray` method which returns the array of attributes that should be converted to JSON when the resource is returned as a response from a route or controller method.

Note that we can access model properties directly from the `$this` variable. This is because a resource class will automatically proxy property and method access down to the underlying model for convenient access. Once the resource is defined, it may be returned from a route or controller. The resource accepts the underlying model instance via its constructor:

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

Resource Collections

If you are returning a collection of resources or a paginated response, you should use the `collection` method provided by your resource class when creating the resource instance in your route or controller:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Note that this does not allow any addition of custom meta data that may need to be returned with your collection. If you would like to customize the resource collection response, you may create a dedicated resource to represent the collection:

```
php artisan make:resource UserCollection
```

Once the resource collection class has been generated, you may easily define any meta data that should be included with the response:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

After defining your resource collection, it may be returned from a route or controller:

```php
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Preserving Collection Keys

When returning a resource collection from a route, Laravel resets the collection's keys so that they are in numerical order. However, you may add a `preserveKeys` property to your resource class indicating whether a collection's original keys should be preserved:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

When the `preserveKeys` property is set to `true`, collection keys will be preserved when the collection is returned from a route or controller:

```php
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

Customizing The Underlying Resource Class

Typically, the `$this->collection` property of a resource collection is automatically populated with the result of mapping each item of the collection to its singular resource class. The singular resource class is assumed to be the collection's class name without the trailing `Collection` portion of the class name. In addition, depending on your personal preference, the singular resource class may or may not be suffixed with `Resource`.

For example, `UserCollection` will attempt to map the given user instances into the `UserResource` resource. To customize this behavior, you may override the `$collects` property of your resource collection:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * The resource that this resource collects.
     *
     * @var string
     */
    public $collects = Member::class;
}
```

Writing Resources

{tip} If you have not read the concept overview, you are highly encouraged to do so before proceeding with this documentation.

In essence, resources are simple. They only need to transform a given model into an array. So, each resource contains a `toArray` method which translates your model's attributes into an API friendly array that can be returned from your application's routes or controllers:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Once a resource has been defined, it may be returned directly from a route or controller:

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

Relationships

If you would like to include related resources in your response, you may add them to the array returned by your resource's `toArray` method. In this example, we will use the `PostResource` resource's `collection` method to add the user's blog posts to the resource response:

```
use App\Http\Resources\PostResource;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

{tip} If you would like to include relationships only when they have already been loaded, check out the documentation on conditional relationships.

Resource Collections

While resources transform a single model into an array, resource collections transform a

collection of models into an array. However, it is not absolutely necessary to define a resource collection class for each one of your models since all resources provide a `collection` method to generate an "ad-hoc" resource collection on the fly:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

However, if you need to customize the meta data returned with the collection, it is necessary to define your own resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Like singular resources, resource collections may be returned directly from routes or controllers:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Data Wrapping

By default, your outermost resource is wrapped in a `data` key when the resource response is converted to JSON. So, for example, a typical resource collection response looks like the following:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ]
}
```

If you would like to use a custom key instead of `data`, you may define a `$wrap` attribute on the resource class:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * The "data" wrapper that should be applied.
     *
     * @var string
     */
    public static $wrap = 'user';
}
```

If you would like to disable the wrapping of the outermost resource, you should invoke the `withoutWrapping` method on the base `Illuminate\Http\Resources\Json\JsonResource` class. Typically, you should call this method from your `AppServiceProvider` or another service provider that is loaded on every request to your application:

```php
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

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

{note} The `withoutWrapping` method only affects the outermost response and will not remove `data` keys that you manually add to your own resource collections.

Wrapping Nested Resources

You have total freedom to determine how your resource's relationships are wrapped. If you would like all resource collections to be wrapped in a `data` key, regardless of their nesting, you should define a resource collection class for each resource and return the collection within a `data` key.

You may be wondering if this will cause your outermost resource to be wrapped in two `data` keys. Don't worry, Laravel will never let your resources be accidentally double-wrapped, so you don't have to be concerned about the nesting level of the resource collection you are

transforming:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

Data Wrapping And Pagination

When returning paginated collections via a resource response, Laravel will wrap your resource data in a `data` key even if the `withoutWrapping` method has been called. This is because paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```


Pagination

You may pass a Laravel paginator instance to the `collection` method of a resource or to a custom resource collection:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Conditional Attributes

Sometimes you may wish to only include an attribute in a resource response if a given condition is met. For example, you may wish to only include a value if the current user is an "administrator". Laravel provides a variety of helper methods to assist you in this situation. The `when` method may be used to conditionally add an attribute to a resource response:

```
use Illuminate\Support\Facades\Auth;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when(Auth::user()->isAdmin(), 'secret-
value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, the `secret` key will only be returned in the final resource response if the authenticated user's `isAdmin` method returns `true`. If the method returns `false`, the `secret` key will be removed from the resource response before it is sent to the client. The `when` method allows you to expressively define your resources without resorting to conditional statements when building the array.

The `when` method also accepts a closure as its second argument, allowing you to calculate the resulting value only if the given condition is `true`:

```
'secret' => $this->when(Auth::user()->isAdmin(), function () {
    return 'secret-value';
}),
```

Merging Conditional Attributes

Sometimes you may have several attributes that should only be included in the resource response based on the same condition. In this case, you may use the `mergeWhen` method to include the attributes in the response only when the given condition is `true`:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen(Auth::user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

Again, if the given condition is `false`, these attributes will be removed from the resource response before it is sent to the client.

{note} The `mergeWhen` method should not be used within arrays that mix string and numeric keys. Furthermore, it should not be used within arrays with numeric keys that are not ordered sequentially.

Conditional Relationships

In addition to conditionally loading attributes, you may conditionally include relationships on your resource responses based on if the relationship has already been loaded on the model. This allows your controller to decide which relationships should be loaded on the model and your resource can easily include them only when they have actually been loaded. Ultimately, this makes it easier to avoid "N+1" query problems within your resources.

The `whenLoaded` method may be used to conditionally load a relationship. In order to avoid unnecessarily loading relationships, this method accepts the name of the relationship instead of the relationship itself:

```
use App\Http\Resources\PostResource;

/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, if the relationship has not been loaded, the `posts` key will be removed from the resource response before it is sent to the client.

Conditional Pivot Information

In addition to conditionally including relationship information in your resource responses, you may conditionally include data from the intermediate tables of many-to-many relationships using the `whenPivotLoaded` method. The `whenPivotLoaded` method accepts the name of the pivot table as its first argument. The second argument should be a closure that returns the value to be returned if the pivot information is available on the model:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_user', function ()
{
            return $this->pivot->expires_at;
        }),
    ];
}
```

If your relationship is using a custom intermediate table model, you may pass an instance of the intermediate table model as the first argument to the `whenPivotLoaded` method:

```
'expires_at' => $this->whenPivotLoaded(new Membership, function () {
    return $this->pivot->expires_at;
}),
```

If your intermediate table is using an accessor other than `pivot`, you may use the `whenPivotLoadedAs` method:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription',
'role_user', function () {
            return $this->subscription->expires_at;
        }),
    ];
}
```

Adding Meta Data

Some JSON API standards require the addition of meta data to your resource and resource collections responses. This often includes things like `links` to the resource or related resources, or meta data about the resource itself. If you need to return additional meta data about a resource, include it in your `toArray` method. For example, you might include `link` information when transforming a resource collection:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

When returning additional meta data from your resources, you never have to worry about accidentally overriding the `links` or `meta` keys that are automatically added by Laravel when returning paginated responses. Any additional `links` you define will be merged with the links provided by the paginator.

Top Level Meta Data

Sometimes you may wish to only include certain meta data with a resource response if the resource is the outermost resource being returned. Typically, this includes meta information about the response as a whole. To define this meta data, add a `with` method to your resource class. This method should return an array of meta data to be included with the resource response only when the resource is the outermost resource being transformed:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource
array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function with($request)
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}
```

Adding Meta Data When Constructing Resources

You may also add top-level data when constructing resource instances in your route or controller. The `additional` method, which is available on all resources, accepts an array of data that should be added to the resource response:

```
return (new UserCollection(User::all()->load('roles')))
                ->additional(['meta' => [
                    'key' => 'value',
                ]]);
```

Resource Responses

As you have already read, resources may be returned directly from routes and controllers:

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

However, sometimes you may need to customize the outgoing HTTP response before it is sent to the client. There are two ways to accomplish this. First, you may chain the `response` method onto the resource. This method will return an `Illuminate\Http\JsonResponse` instance, giving you full control over the response's headers:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
                ->response()
                ->header('X-Value', 'True');
});
```

Alternatively, you may define a `withResponse` method within the resource itself. This method will be called when the resource is returned as the outermost resource in a response:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Illuminate\Http\Response  $response
     * @return void
     */
    public function withResponse($request, $response)
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent: Serialization

- Date Serialization

Introduction

When building APIs using Laravel, you will often need to convert your models and relationships to arrays or JSON. Eloquent includes convenient methods for making these conversions, as well as controlling which attributes are included in the serialized representation of your models.

{tip} For an even more robust way of handling Eloquent model and collection JSON serialization, check out the documentation on Eloquent API resources.

Serializing Models & Collections

Serializing To Arrays

To convert a model and its loaded relationships to an array, you should use the `toArray` method. This method is recursive, so all attributes and all relations (including the relations of relations) will be converted to arrays:

```
use App\Models\User;

$user = User::with('roles')->first();

return $user->toArray();
```

The `attributesToArray` method may be used to convert a model's attributes to an array but not its relationships:

```
$user = User::first();

return $user->attributesToArray();
```

You may also convert entire collections of models to arrays by calling the `toArray` method on the collection instance:

```
$users = User::all();

return $users->toArray();
```

Serializing To JSON

To convert a model to JSON, you should use the `toJson` method. Like `toArray`, the `toJson` method is recursive, so all attributes and relations will be converted to JSON. You may also specify any JSON encoding options that are supported by PHP:

```
use App\Models\User;

$user = User::find(1);

return $user->toJson();

return $user->toJson(JSON_PRETTY_PRINT);
```

Alternatively, you may cast a model or collection to a string, which will automatically call the `toJson` method on the model or collection:

```
return (string) User::find(1);
```

Since models and collections are converted to JSON when cast to a string, you can return Eloquent objects directly from your application's routes or controllers. Laravel will automatically serialize your Eloquent models and collections to JSON when they are returned from routes or controllers:

```
Route::get('users', function () {
    return User::all();
});
```

Relationships

When an Eloquent model is converted to JSON, its loaded relationships will automatically be included as attributes on the JSON object. Also, though Eloquent relationship methods are defined using "camel case" method names, a relationship's JSON attribute will be "snake case".

Hiding Attributes From JSON

Sometimes you may wish to limit the attributes, such as passwords, that are included in your model's array or JSON representation. To do so, add a `$hidden` property to your model. In attributes that are listed in the `$hidden` property's array will not be included in the serialized representation of your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

{tip} To hide relationships, add the relationship's method name to your Eloquent model's `$hidden` property.

Alternatively, you may use the `visible` property to define an "allow list" of attributes that should be included in your model's array and JSON representation. All attributes that are not present in the `$visible` array will be hidden when the model is converted to an array or JSON:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be visible in arrays.
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

Temporarily Modifying Attribute Visibility

If you would like to make some typically hidden attributes visible on a given model instance, you may use the `makeVisible` method. The `makeVisible` method returns the model instance:

```php
return $user->makeVisible('attribute')->toArray();
```

Likewise, if you would like to hide some attributes that are typically visible, you may use the `makeHidden` method.

```php
return $user->makeHidden('attribute')->toArray();
```

Appending Values To JSON

Occasionally, when converting models to arrays or JSON, you may wish to add attributes that do not have a corresponding column in your database. To do so, first define an accessor for the value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Determine if the user is an administrator.
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] === 'yes';
    }
}
```

After creating the accessor, add the attribute name to the `appends` property of your model. Note that attribute names are typically referenced using their "snake case" serialized representation, even though the accessor's PHP method is defined using "camel case":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The accessors to append to the model's array form.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

Once the attribute has been added to the `appends` list, it will be included in both the model's array and JSON representations. Attributes in the `appends` array will also respect the `visible` and `hidden` settings configured on the model.

Appending At Run Time

At runtime, you may instruct a model instance to append additional attributes using the `append` method. Or, you may use the `setAppends` method to override the entire array of appended properties for a given model instance:

```php
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

Date Serialization

Customizing The Default Date Format

You may customize the default serialization format by overriding the `serializeDate` method. This method does not affect how your dates are formatted for storage in the database:

```
/**
 * Prepare a date for array / JSON serialization.
 *
 * @param  \DateTimeInterface  $date
 * @return string
 */
protected function serializeDate(DateTimeInterface $date)
{
    return $date->format('Y-m-d');
}
```

Customizing The Date Format Per Attribute

You may customize the serialization format of individual Eloquent date attributes by specifying the date format in the model's cast declarations:

```
protected $casts = [
    'birthday' => 'date:Y-m-d',
    'joined_at' => 'datetime:Y-m-d H:00',
];
```

Laravel Envoy

Introduction

Laravel Envoy is a tool for executing common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems. However, Windows support is achievable using WSL2.

Installation

First, install Envoy into your project using the Composer package manager:

```
composer require laravel/envoy --dev
```

Once Envoy has been installed, the Envoy binary will be available in your application's `vendor/bin` directory:

```
php vendor/bin/envoy
```

Writing Tasks

Defining Tasks

Tasks are the basic building block of Envoy. Tasks define the shell commands that should execute on your remote servers when the task is invoked. For example, you might define a task that executes the `php artisan queue:restart` command on all of your application's queue worker servers.

All of your Envoy tasks should be defined in an `Envoy.blade.php` file at the root of your application. Here's an example to get you started:

```
@servers(['web' => ['user@192.168.1.1'], 'workers' =>
['user@192.168.1.2']])

@task('restart-queues', ['on' => 'workers'])
    cd /home/user/example.com
    php artisan queue:restart
@endtask
```

As you can see, an array of `@servers` is defined at the top of the file, allowing you to reference these servers via the `on` option of your task declarations. The `@servers` declaration should always be placed on a single line. Within your `@task` declarations, you should place the shell commands that should execute on your servers when the task is invoked.

Local Tasks

You can force a script to run on your local computer by specifying the server's IP address as `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Importing Envoy Tasks

Using the `@import` directive, you may import other Envoy files so their stories and tasks are

added to yours. After the files have been imported, you may execute the tasks they contain as if they were defined in your own Envoy file:

```
@import('vendor/package/Envoy.blade.php')
```

Multiple Servers

Envoy allows you to easily run a task across multiple servers. First, add additional servers to your `@servers` declaration. Each server should be assigned a unique name. Once you have defined your additional servers you may list each of the servers in the task's `on` array:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Parallel Execution

By default, tasks will be executed on each server serially. In other words, a task will finish running on the first server before proceeding to execute on the second server. If you would like to run a task across multiple servers in parallel, add the `parallel` option to your task declaration:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate --force
@endtask
```

Setup

Sometimes, you may need to execute arbitrary PHP code before running your Envoy tasks. You may use the `@setup` directive to define a block of PHP code that should execute before your tasks:

```
@setup
    $now = new DateTime;
@endsetup
```

If you need to require other PHP files before your task is executed, you may use the `@include` directive at the top of your `Envoy.blade.php` file:

```
@include('vendor/autoload.php')

@task('restart-queues')
    # ...
@endtask
```

Variables

If needed, you may pass arguments to Envoy tasks by specifying them on the command line when invoking Envoy:

```
php vendor/bin/envoy run deploy --branch=master
```

You may access the options within your tasks using Blade's "echo" syntax. You may also define Blade `if` statements and loops within your tasks. For example, let's verify the presence of the `$branch` variable before executing the `git pull` command:

```
@servers(['web' => ['user@192.168.1.1']])

@task('deploy', ['on' => 'web'])
    cd /home/user/example.com

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate --force
@endtask
```

Stories

Stories group a set of tasks under a single, convenient name. For instance, a `deploy` story may run the `update-code` and `install-dependencies` tasks by listing the task names within its definition:

```
@servers(['web' => ['user@192.168.1.1']])

@story('deploy')
    update-code
    install-dependencies
@endstory

@task('update-code')
    cd /home/user/example.com
    git pull origin master
@endtask

@task('install-dependencies')
    cd /home/user/example.com
    composer install
@endtask
```

Once the story has been written, you may invoke it in the same way you would invoke a task:

```
php vendor/bin/envoy run deploy
```

Completion Hooks

When tasks and stories finish, a number of hooks are executed. The hook types supported by Envoy are `@after`, `@error`, `@success`, and `@finished`. All of the code in these hooks is interpreted as PHP and executed locally, not on the remote servers that your tasks interact with.

You may define as many of each of these hooks as you like. They will be executed in the order that they appear in your Envoy script.

`@after`

After each task execution, all of the `@after` hooks registered in your Envoy script will execute. The `@after` hooks receive the name of the task that was executed:

```
@after
    if ($task === 'deploy') {
        // ...
    }
@endafter
```

`@error`

After every task failure (exits with a status code greater than `0`), all of the `@error` hooks registered in your Envoy script will execute. The `@error` hooks receive the name of the task that was executed:

```
@error
    if ($task === 'deploy') {
        // ...
    }
@enderror
```

`@success`

If all tasks have executed without errors, all of the `@success` hooks registered in your Envoy script will execute:

```
@success
    // ...
@endsuccess
```

`@finished`

After all tasks have been executed (regardless of exit status), all of the `@finished` hooks will be executed. The `@finished` hooks receive the status code of the completed task, which may be `null` or an `integer` greater than or equal to `0`:

```
@finished
    if ($exitCode > 0) {
        // There were errors in one of the tasks...
    }
@endfinished
```

Running Tasks

To run a task or story that is defined in your application's `Envoy.blade.php` file, execute Envoy's `run` command, passing the name of the task or story you would like to execute. Envoy will execute the task and display the output from your remote servers as the task is running:

```
php vendor/bin/envoy run deploy
```

Confirming Task Execution

If you would like to be prompted for confirmation before running a given task on your servers, you should add the `confirm` directive to your task declaration. This option is particularly useful for destructive operations:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Notifications

Slack

Envoy supports sending notifications to Slack after each task is executed. The `@slack` directive accepts a Slack hook URL and a channel / user name. You may retrieve your webhook URL by creating an "Incoming WebHooks" integration in your Slack control panel.

You should pass the entire webhook URL as the first argument given to the `@slack` directive. The second argument given to the `@slack` directive should be a channel name (`#channel`) or a user name (`@user`):

```
@finished
    @slack('webhook-url', '#bots')
@endfinished
```

By default, Envoy notifications will send a message to the notification channel describing the task that was executed. However, you may overwrite this message with your own custom message by passing a third argument to the `@slack` directive:

```
@finished
    @slack('webhook-url', '#bots', 'Hello, Slack.')
@endfinished
```

Discord

Envoy also supports sending notifications to Discord after each task is executed. The `@discord` directive accepts a Discord hook URL and a message. You may retrieve your webhook URL by creating a "Webhook" in your Server Settings and choosing which channel the webhook should post to. You should pass the entire Webhook URL into the `@discord` directive:

```
@finished
    @discord('discord-webhook-url')
@endfinished
```

Telegram

Envoy also supports sending notifications to Telegram after each task is executed. The `@telegram` directive accepts a Telegram Bot ID and a Chat ID. You may retrieve your Bot ID by creating a new bot using BotFather. You can retrieve a valid Chat ID using @username_to_id_bot. You should pass the entire Bot ID and Chat ID into the `@telegram` directive:

```
@finished
    @telegram('bot-id','chat-id')
@endfinished
```

Microsoft Teams

Envoy also supports sending notifications to Microsoft Teams after each task is executed. The `@microsoftTeams` directive accepts a Teams Webhook (required), a message, theme color (success, info, warning, error), and an array of options. You may retrieve your Teams Webook by creating a new incoming webhook. The Teams API has many other attributes to customize your message box like title, summary, and sections. You can find more information on the Microsoft Teams documentation. You should pass the entire Webhook URL into the `@microsoftTeams` directive:

```
@finished
    @microsoftTeams('webhook-url')
@endfinished
```

Events

Introduction

Laravel's events provide a simple observer pattern implementation, allowing you to subscribe and listen for various events that occur within your application. Event classes are typically stored in the `app/Events` directory, while their listeners are stored in `app/Listeners`. Don't worry if you don't see these directories in your application as they will be created for you as you generate events and listeners using Artisan console commands.

Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other. For example, you may wish to send a Slack notification to your user each time an order has shipped. Instead of coupling your order processing code to your Slack notification code, you can raise an `App\Events\OrderShipped` event which a listener can receive and use to dispatch a Slack notification.

Registering Events & Listeners

The `App\Providers\EventServiceProvider` included with your Laravel application provides a convenient place to register all of your application's event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). You may add as many events to this array as your application requires. For example, let's add an `OrderShipped` event:

```
use App\Events\OrderShipped;
use App\Listeners\SendShipmentNotification;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    OrderShipped::class => [
        SendShipmentNotification::class,
    ],
];
```

{tip} The `event:list` command may be used to display a list of all events and listeners registered by your application.

Generating Events & Listeners

Of course, manually creating the files for each event and listener is cumbersome. Instead, add listeners and events to your `EventServiceProvider` and use the `event:generate` Artisan command. This command will generate any events or listeners that are listed in your `EventServiceProvider` that do not already exist:

```
php artisan event:generate
```

Alternatively, you may use the `make:event` and `make:listener` Artisan commands to generate individual events and listeners:

```
php artisan make:event PodcastProcessed

php artisan make:listener SendPodcastNotification --
event=PodcastProcessed
```

Manually Registering Events

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register class or closure based event listeners manually in the `boot` method of your `EventServiceProvider`:

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

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    Event::listen(
        PodcastProcessed::class,
        [SendPodcastNotification::class, 'handle']
    );

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

Queueable Anonymous Event Listeners

When registering closure based event listeners manually, you may wrap the listener closure within the `Illuminate\Events\queueable` function to instruct Laravel to execute the listener using the queue:

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

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        //
    }));
}
```

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

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

If you would like to handle anonymous queued listener failures, you may provide a closure to the `catch` method while defining the `queueable` listener. This closure will receive the event instance and the `Throwable` instance that caused the listener's failure:

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

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

Wildcard Event Listeners

You may even register listeners using the `*` as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the event name as their first argument and the entire event data array as their second argument:

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

Event Discovery

Instead of registering events and listeners manually in the `$listen` array of the `EventServiceProvider`, you can enable automatic event discovery. When event discovery is enabled, Laravel will automatically find and register your events and listeners by scanning your application's `Listeners` directory. In addition, any explicitly defined events listed in the `EventServiceProvider` will still be registered.

Laravel finds event listeners by scanning the listener classes using PHP's reflection services. When Laravel finds any listener class method that begins with `handle`, Laravel will register those methods as event listeners for the event that is type-hinted in the method's signature:

```
use App\Events\PodcastProcessed;

class SendPodcastNotification
{
    /**
     * Handle the given event.
     *
     * @param  \App\Events\PodcastProcessed  $event
     * @return void
     */
    public function handle(PodcastProcessed $event)
    {
        //
    }
}
```

Event discovery is disabled by default, but you can enable it by overriding the `shouldDiscoverEvents` method of your application's `EventServiceProvider`:

```
/**
 * Determine if events and listeners should be automatically discovered.
 *
 * @return bool
 */
public function shouldDiscoverEvents()
{
    return true;
}
```

By default, all listeners within your application's `app/Listeners` directory will be scanned. If you would like to define additional directories to scan, you may override the `discoverEventsWithin` method in your `EventServiceProvider`:

```
/**
 * Get the listener directories that should be used to discover events.
 *
 * @return array
 */
protected function discoverEventsWithin()
{
    return [
        $this->app->path('Listeners'),
    ];
}
```

Event Discovery In Production

In production, it is not efficient for the framework to scan all of your listeners on every request. Therefore, during your deployment process, you should run the `event:cache` Artisan command to cache a manifest of all of your application's events and listeners. This manifest will be used by the framework to speed up the event registration process. The `event:clear` command may be used to destroy the cache.

Defining Events

An event class is essentially a data container which holds the information related to the event. For example, let's assume an `App\Events\OrderShipped` event receives an Eloquent ORM object:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;

    /**
     * Create a new event instance.
     *
     * @param  \App\Models\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}
```

As you can see, this event class contains no logic. It is a container for the `App\Models\Order` instance that was purchased. The `SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using

PHP's `serialize` function, such as when utilizing queued listeners.

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive event instances in their `handle` method. The `event:generate` and `make:listener` Artisan commands will automatically import the proper event class and type-hint the event on the `handle` method. Within the `handle` method, you may perform any actions necessary to respond to the event:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // Access the order using $event->order...
    }
}
```

{tip} Your event listeners may also type-hint any dependencies they need on their constructors. All event listeners are resolved via the Laravel service container, so dependencies will be injected automatically.

Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so by returning `false` from your listener's `handle` method.

Queued Event Listeners

Queueing listeners can be beneficial if your listener is going to perform a slow task such as sending an email or making an HTTP request. Before using queued listeners, make sure to configure your queue and start a queue worker on your server or local development environment.

To specify that a listener should be queued, add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` and `make:listener` Artisan commands already have this interface imported into the current namespace so you can use it immediately:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}
```

That's it! Now, when an event handled by this listener is dispatched, the listener will automatically be queued by the event dispatcher using Laravel's queue system. If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has finished processing.

Customizing The Queue Connection & Queue Name

If you would like to customize the queue connection, queue name, or queue delay time of an event listener, you may define the `$connection`, `$queue`, or `$delay` properties on your listener class:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * The name of the connection the job should be sent to.
     *
     * @var string|null
     */
    public $connection = 'sqs';

    /**
     * The name of the queue the job should be sent to.
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * The time (seconds) before the job should be processed.
     *
     * @var int
     */
    public $delay = 60;
}
```

If you would like to define the listener's queue connection or queue name at runtime, you may define `viaConnection` or `viaQueue` methods on the listener:

```
/**
 * Get the name of the listener's queue connection.
 *
 * @return string
 */
public function viaConnection()
{
    return 'sqs';
}

/**
 * Get the name of the listener's queue.
 *
 * @return string
 */
public function viaQueue()
{
    return 'listeners';
}
```

Conditionally Queueing Listeners

Sometimes, you may need to determine whether a listener should be queued based on some data that are only available at runtime. To accomplish this, a `shouldQueue` method may be added to a listener to determine whether the listener should be queued. If the `shouldQueue` method returns `false`, the listener will not be executed:

```php
<?php

namespace App\Listeners;

use App\Events\OrderCreated;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * Reward a gift card to the customer.
     *
     * @param  \App\Events\OrderCreated  $event
     * @return void
     */
    public function handle(OrderCreated $event)
    {
        //
    }

    /**
     * Determine whether the listener should be queued.
     *
     * @param  \App\Events\OrderCreated  $event
     * @return bool
     */
    public function shouldQueue(OrderCreated $event)
    {
        return $event->order->subtotal >= 5000;
    }
}
```

Manually Interacting With The Queue

If you need to manually access the listener's underlying queue job's `delete` and `release` methods, you may do so using the `Illuminate\Queue\InteractsWithQueue` trait. This trait is imported by default on generated listeners and provides access to these methods:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

Queued Event Listeners & Database Transactions

When queued listeners are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your listener depends on these models, unexpected errors can occur when the job that dispatches the queued listener is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued listener should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the listener class:

```php
<?php

namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public $afterCommit = true;
}
```

{tip} To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Handling Failed Jobs

Sometimes your queued event listeners may fail. If queued listener exceeds the maximum number of attempts as defined by your queue worker, the `failed` method will be called on your listener. The `failed` method receives the event instance and the `Throwable` that caused the failure:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        //
    }

    /**
     * Handle a job failure.
     *
     * @param  \App\Events\OrderShipped  $event
     * @param  \Throwable  $exception
     * @return void
     */
    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

Specifying Queued Listener Maximum Attempts

If one of your queued listeners is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a listener may be attempted.

You may define `$tries` property on your listener class to specify how many times the

listener may be attempted before it is considered to have failed:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * The number of times the queued listener may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

As an alternative to defining how many times a listener may be attempted before it fails, you may define a time at which the listener should no longer be attempted. This allows a listener to be attempted any number of times within a given time frame. To define the time at which a listener should no longer be attempted, add a `retryUntil` method to your listener class. This method should return a `DateTime` instance:

```
/**
 * Determine the time at which the listener should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addMinutes(5);
}
```

Dispatching Events

To dispatch an event, you may call the static `dispatch` method on the event. This method is made available on the event by the `Illuminate\Foundation\Events\Dispatchable` trait. Any arguments passed to the `dispatch` method will be passed to the event's constructor:

```
<?php

namespace App\Http\Controllers;

use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\Request;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $order = Order::findOrFail($request->order_id);

        // Order shipment logic...

        OrderShipped::dispatch($order);
    }
}
```

{tip} When testing, it can be helpful to assert that certain events were dispatched without actually triggering their listeners. Laravel's built-in testing helpers makes it a cinch.

Event Subscribers

Writing Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the subscriber class itself, allowing you to define several event handlers within a single class. Subscribers should define a `subscribe` method, which will be passed an event dispatcher instance. You may call the `listen` method on the given dispatcher to register event listeners:

```php
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout($event) {}

    /**
     * Register the listeners for the subscriber.
     *
     * @param  \Illuminate\Events\Dispatcher  $events
     * @return void
     */
    public function subscribe($events)
    {
        $events->listen(
            Login::class,
            [UserEventSubscriber::class, 'handleUserLogin']
        );

        $events->listen(
            Logout::class,
            [UserEventSubscriber::class, 'handleUserLogout']
        );
    }
}
```

If your event listener methods are defined within the subscriber itself, you may find it more convenient to return an array of events and method names from the subscriber's `subscribe` method. Laravel will automatically determine the subscriber's class name when registering the event listeners:

```php
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout($event) {}

    /**
     * Register the listeners for the subscriber.
     *
     * @param  \Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        return [
            Login::class => 'handleUserLogin',
            Logout::class => 'handleUserLogout',
        ];
    }
}
```

Registering Event Subscribers

After writing the subscriber, you are ready to register it with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventSubscriber` to the list:

```php
<?php

namespace App\Providers;

use App\Listeners\UserEventSubscriber;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        UserEventSubscriber::class,
    ];
}
```

Facades

Introduction

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

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

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

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

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

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

Helper Functions

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

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

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

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

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

When To Use Facades

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

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

Facades Vs. Dependency Injection

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

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

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

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

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

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

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

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

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

Facades Vs. Helper Functions

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

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

return view('profile');
```

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

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

Under the hood, the `cache` helper is going to call the `get` method on the class underlying

the `Cache` facade. So, even though we are using the helper function, we can write the following test to verify that the method was called with the argument we expected:

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

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

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

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

How Facades Work

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

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

```
<?php

namespace App\Http\Controllers;

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

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

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

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

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

static method `get`:

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

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

Real-Time Facades

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

```
<?php

namespace App\Models;

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

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

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

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

```php
<?php

namespace App\Models;

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

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

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

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

```php
<?php

namespace Tests\Feature;

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

class PodcastTest extends TestCase
{
    use RefreshDatabase;

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

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

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

Facade Class Reference

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

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Contracts\Console\Kernel	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Contracts\Auth\Guard	`auth.driver`
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\CacheManager	`cache`
Cache (Instance)	Illuminate\Cache\Repository	`cache.store`
Config	Illuminate\Config\Repository	`config`
Cookie	Illuminate\Cookie\CookieJar	`cookie`
Crypt	Illuminate\Encryption\Encrypter	`encrypter`
Date	Illuminate\Support\DateFactory	`date`
DB	Illuminate\Database\DatabaseManager	`db`
DB (Instance)	Illuminate\Database\Connection	`db.connection`
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Gate	Illuminate\Contracts\Auth\Access\Gate	

Facade	Class	Service Container Binding
Hash	Illuminate\Contracts\Hashing\Hasher	`hash`
Http	Illuminate\Http\Client\Factory	
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\LogManager	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	`auth.password`
Password (Instance)	Illuminate\Auth\Passwords\PasswordBroker	`auth.password.broker`
Queue	Illuminate\Queue\QueueManager	`queue`
Queue (Instance)	Illuminate\Contracts\Queue\Queue	`queue.connection`
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\RedisManager	`redis`
Redis (Instance)	Illuminate\Redis\Connections\Connection	`redis.connection`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (Instance)	Illuminate\Http\Response	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Builder	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	`session.store`
Storage	Illuminate\Filesystem\FilesystemManager	`filesystem`
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	`filesystem.disk`

Facade	Class	Service Container Binding
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	

Laravel Fortify

Introduction

Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more. After installing Fortify, you may run the `route:list` Artisan command to see the routes that Fortify has registered.

Since Fortify does not provide its own user interface, it is meant to be paired with your own user interface which makes requests to the routes it registers. We will discuss exactly how to make requests to these routes in the remainder of this documentation.

{tip} Remember, Fortify is a package that is meant to give you a head start implementing Laravel's authentication features. **You are not required to use it.** You are always free to manually interact with Laravel's authentication services by following the documentation available in the authentication, password reset, and email verification documentation.

What Is Fortify?

As mentioned previously, Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more.

You are not required to use Fortify in order to use Laravel's authentication features. You are always free to manually interact with Laravel's authentication services by following the documentation available in the authentication, password reset, and email verification documentation.

If you are new to Laravel, you may wish to explore the Laravel Breeze application starter kit before attempting to use Laravel Fortify. Laravel Breeze provides an authentication scaffolding for your application that includes a user interface built with Tailwind CSS. Unlike Fortify, Breeze publishes its routes and controllers directly into your application. This allows you to study and get comfortable with Laravel's authentication features before allowing Laravel Fortify to implement these features for you.

Laravel Fortify essentially takes the routes and controllers of Laravel Breeze and offers them as a package that does not include a user interface. This allows you to still quickly scaffold the backend implementation of your application's authentication layer without being tied to any particular frontend opinions.

When Should I Use Fortify?

You may be wondering when it is appropriate to use Laravel Fortify. First, if you are using one of Laravel's application starter kits, you do not need to install Laravel Fortify since all of Laravel's application starter kits already provide a full authentication implementation.

If you are not using an application starter kit and your application needs authentication features, you have two options: manually implement your application's authentication features or use Laravel Fortify to provide the backend implementation of these features.

If you choose to install Fortify, your user interface will make requests to Fortify's authentication routes that are detailed in this documentation in order to authenticate and register users.

If you choose to manually interact with Laravel's authentication services instead of using Fortify, you may do so by following the documentation available in the authentication, password reset, and email verification documentation.

Laravel Fortify & Laravel Sanctum

Some developers become confused regarding the difference between Laravel Sanctum and Laravel Fortify. Because the two packages solve two different but related problems, Laravel Fortify and Laravel Sanctum are not mutually exclusive or competing packages.

Laravel Sanctum is only concerned with managing API tokens and authenticating existing users using session cookies or tokens. Sanctum does not provide any routes that handle user registration, password reset, etc.

If you are attempting to manually build the authentication layer for an application that offers an API or serves as the backend for a single-page application, it is entirely possible that you will utilize both Laravel Fortify (for user registration, password reset, etc.) and Laravel Sanctum (API token management, session authentication).

Installation

To get started, install Fortify using the Composer package manager:

```
composer require laravel/fortify
```

Next, publish Fortify's resources using the `vendor:publish` command:

```
php artisan vendor:publish --
provider="Laravel\Fortify\FortifyServiceProvider"
```

This command will publish Fortify's actions to your `app/Actions` directory, which will be created if it does not exist. In addition, Fortify's configuration file and migrations will be published.

Next, you should migrate your database:

```
php artisan migrate
```

The Fortify Service Provider

The `vendor:publish` command discussed above will also publish the `App\Providers\FortifyServiceProvider` class. You should ensure this class is registered within the `providers` array of your application's `config/app.php` configuration file.

The Fortify service provider registers the actions that Fortify published and instructs Fortify to use them when their respective tasks are executed by Fortify.

Fortify Features

The `fortify` configuration file contains a `features` configuration array. This array defines which backend routes / features Fortify will expose by default. If you are not using Fortify in

combination with Laravel Jetstream, we recommend that you only enable the following features, which are the basic authentication features provided by most Laravel applications:

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
],
```

Disabling Views

By default, Fortify defines routes that are intended to return views, such as a login screen or registration screen. However, if you are building a JavaScript driven single-page application, you may not need these routes. For that reason, you may disable these routes entirely by setting the `views` configuration value within your application's `config/fortify.php` configuration file to `false`:

```
'views' => false,
```

Disabling Views & Password Reset

If you choose to disable Fortify's views and you will be implementing password reset features for your application, you should still define a route named `password.reset` that is responsible for displaying your application's "reset password" view. This is necessary because Laravel's `Illuminate\Auth\Notifications\ResetPassword` notification will generate the password reset URL via the `password.reset` named route.

Authentication

To get started, we need to instruct Fortify how to return our "login" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an application starter kit.

All of the authentication view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class. Fortify will take care of defining the `/login` route that returns this view:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::loginView(function () {
        return view('auth.login');
    });

    // ...
}
```

Your login template should include a form that makes a POST request to `/login`. The `/login` endpoint expects a string `email` / `username` and a `password`. The name of the email / username field should match the `username` value within the `config/fortify.php` configuration file. In addition, a boolean `remember` field may be provided to indicate that the user would like to use the "remember me" functionality provided by Laravel.

If the login attempt is successful, Fortify will redirect you to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the login screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with the

422 HTTP response.

Customizing User Authentication

Fortify will automatically retrieve and authenticate the user based on the provided credentials and the authentication guard that is configured for your application. However, you may sometimes wish to have full customization over how login credentials are authenticated and users are retrieved. Thankfully, Fortify allows you to easily accomplish this using the `Fortify::authenticateUsing` method.

This method accepts a closure which receives the incoming HTTP request. The closure is responsible for validating the login credentials attached to the request and returning the associated user instance. If the credentials are invalid or no user can be found, `null` or `false` should be returned by the closure. Typically, this method should be called from the `boot` method of your `FortifyServiceProvider`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::authenticateUsing(function (Request $request) {
        $user = User::where('email', $request->email)->first();

        if ($user &&
            Hash::check($request->password, $user->password)) {
            return $user;
        }
    });

    // ...
}
```

Authentication Guard

You may customize the authentication guard used by Fortify within your application's `fortify` configuration file. However, you should ensure that the configured guard is an implementation of `Illuminate\Contracts\Auth\StatefulGuard`. If you are attempting to use Laravel Fortify to authenticate an SPA, you should use Laravel's default `web` guard in combination with Laravel Sanctum.

Customizing The Authentication Pipeline

Laravel Fortify authenticates login requests through a pipeline of invokable classes. If you would like, you may define a custom pipeline of classes that login requests should be piped through. Each class should have an `__invoke` method which receives the incoming `Illuminate\Http\Request` instance and, like middleware, a `$next` variable that is invoked in order to pass the request to the next class in the pipeline.

To define your custom pipeline, you may use the `Fortify::authenticateThrough` method. This method accepts a closure which should return the array of classes to pipe the login request through. Typically, this method should be called from the `boot` method of your `App\Providers\FortifyServiceProvider` class.

The example below contains the default pipeline definition that you may use as a starting point when making your own modifications:

```
use Laravel\Fortify\Actions\AttemptToAuthenticate;
use Laravel\Fortify\Actions\EnsureLoginIsNotThrottled;
use Laravel\Fortify\Actions\PrepareAuthenticatedSession;
use Laravel\Fortify\Actions\RedirectIfTwoFactorAuthenticatable;
use Laravel\Fortify\Fortify;
use Illuminate\Http\Request;

Fortify::authenticateThrough(function (Request $request) {
    return array_filter([
            config('fortify.limiters.login') ? null :
EnsureLoginIsNotThrottled::class,
            Features::enabled(Features::twoFactorAuthentication()) ?
RedirectIfTwoFactorAuthenticatable::class : null,
            AttemptToAuthenticate::class,
            PrepareAuthenticatedSession::class,
    ]);
});
```

Customizing Redirects

If the login attempt is successful, Fortify will redirect you to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned. After a user logs out of the application, the user will be redirected to the `/` URI.

If you need advanced customization of this behavior, you may bind implementations of the `LoginResponse` and `LogoutResponse` contracts into the Laravel service container. Typically, this should be done within the `register` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Contracts\LogoutResponse;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->app->instance(LogoutResponse::class, new class implements
LogoutResponse {
        public function toResponse($request)
        {
            return redirect('/');
        }
    });
}
```

Two Factor Authentication

When Fortify's two factor authentication feature is enabled, the user is required to input a six digit numeric token during the authentication process. This token is generated using a time-based one-time password (TOTP) that can be retrieved from any TOTP compatible mobile authentication application such as Google Authenticator.

Before getting started, you should first ensure that your application's `App\Models\User` model uses the `Laravel\Fortify\TwoFactorAuthenticatable` trait:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Fortify\TwoFactorAuthenticatable;

class User extends Authenticatable
{
    use Notifiable, TwoFactorAuthenticatable;
}
```

Next, you should build a screen within your application where users can manage their two factor authentication settings. This screen should allow the user to enable and disable two factor authentication, as well as regenerate their two factor authentication recovery codes.

By default, the `features` array of the `fortify` configuration file instructs Fortify's two factor authentication settings to require password confirmation before modification. Therefore, your application should implement Fortify's password confirmation feature before continuing.

Enabling Two Factor Authentication

To enable two factor authentication, your application should make a POST request to the `/user/two-factor-authentication` endpoint defined by Fortify. If the request is successful, the user will be redirected back to the previous URL and the `status` session variable will be set to `two-factor-authentication-enabled`. You may detect this `status` session variable within your templates to display the appropriate success message.

If the request was an XHR request, `200` HTTP response will be returned:

```
@if (session('status') == 'two-factor-authentication-enabled')
    <div class="mb-4 font-medium text-sm text-green-600">
        Two factor authentication has been enabled.
    </div>
@endif
```

Next, you should display the two factor authentication QR code for the user to scan into their authenticator application. If you are using Blade to render your application's frontend, you may retrieve the QR code SVG using the `twoFactorQrCodeSvg` method available on the user instance:

```
$request->user()->twoFactorQrCodeSvg();
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-qr-code` endpoint to retrieve the user's two factor authentication QR code. This endpoint will return a JSON object containing an `svg` key.

Displaying The Recovery Codes

You should also display the user's two factor recovery codes. These recovery codes allow the user to authenticate if they lose access to their mobile device. If you are using Blade to render your application's frontend, you may access the recovery codes via the authenticated user instance:

```
(array) $request->user()->recoveryCodes()
```

If you are building a JavaScript powered frontend, you may make an XHR GET request to the `/user/two-factor-recovery-codes` endpoint. This endpoint will return a JSON array containing the user's recovery codes.

To regenerate the user's recovery codes, your application should make a POST request to the `/user/two-factor-recovery-codes` endpoint.

Authenticating With Two Factor Authentication

During the authentication process, Fortify will automatically redirect the user to your application's two factor authentication challenge screen. However, if your application is making an XHR login request, the JSON response returned after a successful authentication attempt will contain a JSON object that has a `two_factor` boolean property. You should inspect this value to know whether you should redirect to your application's two factor authentication challenge screen.

To begin implementing two factor authentication functionality, we need to instruct Fortify how to return our two factor authentication challenge view. All of Fortify's authentication view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::twoFactorChallengeView(function () {
        return view('auth.two-factor-challenge');
    });

    // ...
}
```

Fortify will take care of defining the `/two-factor-challenge` route that returns this view. Your `two-factor-challenge` template should include a form that makes a POST request to the `/two-factor-challenge` endpoint. The `/two-factor-challenge` action expects a `code` field that contains a valid TOTP token or a `recovery_code` field that contains one of the user's recovery codes.

If the login attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 204 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the login screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422

HTTP response.

Disabling Two Factor Authentication

To disable two factor authentication, your application should make a DELETE request to the `/user/two-factor-authentication` endpoint. Remember, Fortify's two factor authentication endpoints require password confirmation prior to being called.

Registration

To begin implementing our application's registration functionality, we need to instruct Fortify how to return our "register" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an application starter kit.

All of the Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::registerView(function () {
        return view('auth.register');
    });

    // ...
}
```

Fortify will take care of defining the `/register` route that returns this view. Your `register` template should include a form that makes a POST request to the `/register` endpoint defined by Fortify.

The `/register` endpoint expects a string `name`, string email address / username, `password`, and `password_confirmation` fields. The name of the email / username field should match the `username` configuration value defined within your application's `fortify` configuration file.

If the registration attempt is successful, Fortify will redirect the user to the URI configured via the `home` configuration option within your application's `fortify` configuration file. If the login request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the registration screen and the validation errors will be available to you via the shared `$errors` Blade template

variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Registration

The user validation and creation process may be customized by modifying the `App\Actions\Fortify\CreateNewUser` action that was generated when you installed Laravel Fortify.

Password Reset

Requesting A Password Reset Link

To begin implementing our application's password reset functionality, we need to instruct Fortify how to return our "forgot password" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an application starter kit.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::requestPasswordResetLinkView(function () {
        return view('auth.forgot-password');
    });

    // ...
}
```

Fortify will take care of defining the `/forgot-password` endpoint that returns this view. Your `forgot-password` template should include a form that makes a POST request to the `/forgot-password` endpoint.

The `/forgot-password` endpoint expects a string `email` field. The name of this field / database column should match the `email` configuration value within your application's `fortify` configuration file.

Handling The Password Reset Link Request Response

If the password reset link request was successful, Fortify will redirect the user back to the `/forgot-password` endpoint and send an email to the user with a secure link they can use to reset their password. If the request was an XHR request, a 200 HTTP response will be returned.

After being redirected back to the `/forgot-password` endpoint after a successful request, the `status` session variable may be used to display the status of the password reset link request attempt. The value of this session variable will match one of the translation strings defined within your application's `passwords` language file:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was not successful, the user will be redirected back to the request password reset link screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Resetting The Password

To finish implementing our application's password reset functionality, we need to instruct Fortify how to return our "reset password" view.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::resetPasswordView(function ($request) {
        return view('auth.reset-password', ['request' => $request]);
    });

    // ...
}
```

Fortify will take care of defining the route to display this view. Your `reset-password` template should include a form that makes a POST request to `/reset-password`.

The `/reset-password` endpoint expects a string `email` field, a `password` field, a `password_confirmation` field, and a hidden field named `token` that contains the value of `request()->route('token')`. The name of the "email" field / database column should match the `email` configuration value defined within your application's `fortify` configuration file.

Handling The Password Reset Response

If the password reset request was successful, Fortify will redirect back to the `/login` route so that the user can log in with their new password. In addition, a `status` session variable will be set so that you may display the successful status of the reset on your login screen:

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

If the request was an XHR request, a 200 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the reset password screen and the validation errors will be available to you via the shared `$errors` Blade

template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Customizing Password Resets

The password reset process may be customized by modifying the `App\Actions\ResetUserPassword` action that was generated when you installed Laravel Fortify.

Email Verification

After registration, you may wish for users to verify their email address before they continue accessing your application. To get started, ensure the `emailVerification` feature is enabled in your `fortify` configuration file's `features` array. Next, you should ensure that your `App\Models\User` class implements the `Illuminate\Contracts\Auth\MustVerifyEmail` interface.

Once these two setup steps have been completed, newly registered users will receive an email prompting them to verify their email address ownership. However, we need to inform Fortify how to display the email verification screen which informs the user that they need to go click the verification link in the email.

All of Fortify's view's rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::verifyEmailView(function () {
        return view('auth.verify-email');
    });

    // ...
}
```

Fortify will take care of defining the route that displays this view when a user is redirected to the `/email/verify` endpoint by Laravel's built-in `verified` middleware.

Your `verify-email` template should include an informational message instructing the user to click the email verification link that was sent to their email address.

Resending Email Verification Links

If you wish, you may add a button to your application's `verify-email` template that triggers a POST request to the `/email/verification-notification` endpoint. When this endpoint receives a request, a new verification email link will be emailed to the user, allowing the user to get a new verification link if the previous one was accidentally deleted or lost.

If the request to resend the verification link email was successful, Fortify will redirect the user back to the `/email/verify` endpoint with a `status` session variable, allowing you to display an informational message to the user informing them the operation was successful. If the request was an XHR request, a 202 HTTP response will be returned:

```
@if (session('status') == 'verification-link-sent')
    <div class="mb-4 font-medium text-sm text-green-600">
        A new email verification link has been emailed to you!
    </div>
@endif
```

Protecting Routes

To specify that a route or group of routes requires that the user has verified their email address, you should attach Laravel's built-in `verified` middleware to the route. This middleware is registered within your application's `App\Http\Kernel` class:

```
Route::get('/dashboard', function () {
    // ...
})->middleware(['verified']);
```

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed. Typically, these routes are protected by Laravel's built-in `password.confirm` middleware.

To begin implementing password confirmation functionality, we need to instruct Fortify how to return our application's "password confirmation" view. Remember, Fortify is a headless authentication library. If you would like a frontend implementation of Laravel's authentication features that are already completed for you, you should use an application starter kit.

All of Fortify's view rendering logic may be customized using the appropriate methods available via the `Laravel\Fortify\Fortify` class. Typically, you should call this method from the `boot` method of your application's `App\Providers\FortifyServiceProvider` class:

```
use Laravel\Fortify\Fortify;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Fortify::confirmPasswordView(function () {
        return view('auth.confirm-password');
    });

    // ...
}
```

Fortify will take care of defining the `/user/confirm-password` endpoint that returns this view. Your `confirm-password` template should include a form that makes a POST request to the `/user/confirm-password` endpoint. The `/user/confirm-password` endpoint expects a `password` field that contains the user's current password.

If the password matches the user's current password, Fortify will redirect the user to the route they were attempting to access. If the request was an XHR request, a 201 HTTP response will be returned.

If the request was not successful, the user will be redirected back to the confirm password screen and the validation errors will be available to you via the shared `$errors` Blade template variable. Or, in the case of an XHR request, the validation errors will be returned with a 422 HTTP response.

Laravel Homestead

Introduction

Laravel strives to make the entire PHP development experience delightful, including your local development environment. Laravel Homestead is an official, pre-packaged Vagrant box that provides you a wonderful development environment without requiring you to install PHP, a web server, and any other server software on your local machine.

Vagrant provides a simple, elegant way to manage and provision Virtual Machines. Vagrant boxes are completely disposable. If something goes wrong, you can destroy and re-create the box in minutes!

Homestead runs on any Windows, macOS, or Linux system and includes Nginx, PHP, MySQL, PostgreSQL, Redis, Memcached, Node, and all of the other software you need to develop amazing Laravel applications.

{note} If you are using Windows, you may need to enable hardware virtualization (VT-x). It can usually be enabled via your BIOS. If you are using Hyper-V on a UEFI system you may additionally need to disable Hyper-V in order to access VT-x.

Included Software

- Ubuntu 20.04 - Git - PHP 8.1 - PHP 8.0 - PHP 7.4 - PHP 7.3 - PHP 7.2 - PHP 7.1 - PHP 7.0 - PHP 5.6 - Nginx - MySQL 8.0 - lmm - Sqlite3 - PostgreSQL 13 - Composer - Node (With Yarn, Bower, Grunt, and Gulp) - Redis - Memcached - Beanstalkd - Mailhog - avahi - ngrok - Xdebug - XHProf / Tideways / XHGui - wp-cli

Optional Software

- Apache - Blackfire - Cassandra - Chronograf - CouchDB - Crystal & Lucky Framework - Docker - Elasticsearch - EventStoreDB - Gearman - Go - Grafana - InfluxDB - MariaDB - Meilisearch - MinIO - MongoDB - Neo4j - Oh My Zsh - Open Resty - PM2 - Python - R - RabbitMQ - RVM (Ruby Version Manager) - Solr - TimescaleDB - Trader (PHP extension) - Webdriver & Laravel Dusk Utilities

Installation & Setup

First Steps

Before launching your Homestead environment, you must install Vagrant as well as one of the following supported providers:

- VirtualBox 6.1.x
- Parallels

All of these software packages provide easy-to-use visual installers for all popular operating systems.

To use the Parallels provider, you will need to install Parallels Vagrant plug-in. It is free of charge.

Installing Homestead

You may install Homestead by cloning the Homestead repository onto your host machine. Consider cloning the repository into a `Homestead` folder within your "home" directory, as the Homestead virtual machine will serve as the host to all of your Laravel applications. Throughout this documentation, we will refer to this directory as your "Homestead directory":

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

After cloning the Laravel Homestead repository, you should checkout the `release` branch. This branch always contains the latest stable release of Homestead:

```
cd ~/Homestead

git checkout release
```

Next, execute the `bash init.sh` command from the Homestead directory to create the `Homestead.yaml` configuration file. The `Homestead.yaml` file is where you will configure all of the settings for your Homestead installation. This file will be placed in the Homestead directory:

```
// macOS / Linux...
bash init.sh

// Windows...
init.bat
```

Configuring Homestead

Setting Your Provider

The `provider` key in your `Homestead.yaml` file indicates which Vagrant provider should be used: `virtualbox` or `parallels`:

```
provider: virtualbox
```

{note} If you are using Apple Silicon, you should add `box: laravel/homestead-arm` to your `Homestead.yaml` file. Apple Silicon requires the Parallels provider.

Configuring Shared Folders

The `folders` property of the `Homestead.yaml` file lists all of the folders you wish to share with your Homestead environment. As files within these folders are changed, they will be kept in sync between your local machine and the Homestead virtual environment. You may configure as many shared folders as necessary:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
```

{note} Windows users should not use the `~/` path syntax and instead should use the full path to their project, such as `C:\Users\user\Code\project1`.

You should always map individual applications to their own folder mapping instead of mapping a single large directory that contains all of your applications. When you map a folder, the virtual machine must keep track of all disk IO for *every* file in the folder. You may

experience reduced performance if you have a large number of files in a folder:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
    - map: ~/code/project2
      to: /home/vagrant/project2
```

{note} You should never mount `.` (the current directory) when using Homestead. This causes Vagrant to not map the current folder to `/vagrant` and will break optional features and cause unexpected results while provisioning.

To enable NFS, you may add a `type` option to your folder mapping:

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
      type: "nfs"
```

{note} When using NFS on Windows, you should consider installing the vagrant-winnfsd plug-in. This plug-in will maintain the correct user / group permissions for files and directories within the Homestead virtual machine.

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

```
folders:
    - map: ~/code/project1
      to: /home/vagrant/project1
      type: "rsync"
      options:
          rsync__args: ["--verbose", "--archive", "--delete", "-zz"]
          rsync__exclude: ["node_modules"]
```

Configuring Nginx Sites

Not familiar with Nginx? No problem. Your `Homestead.yaml` file's `sites` property allows

you to easily map a "domain" to a folder on your Homestead environment. A sample site configuration is included in the `Homestead.yaml` file. Again, you may add as many sites to your Homestead environment as necessary. Homestead can serve as a convenient, virtualized environment for every Laravel application you are working on:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
```

If you change the `sites` property after provisioning the Homestead virtual machine, you should execute the `vagrant reload --provision` command in your terminal to update the Nginx configuration on the virtual machine.

{note} Homestead scripts are built to be as idempotent as possible. However, if you are experiencing issues while provisioning you should destroy and rebuild the machine by executing the `vagrant destroy && vagrant up` command.

Hostname Resolution

Homestead publishes hostnames using `mDNS` for automatic host resolution. If you set `hostname: homestead` in your `Homestead.yaml` file, the host will be available at `homestead.local`. macOS, iOS, and Linux desktop distributions include `mDNS` support by default. If you are using Windows, you must install Bonjour Print Services for Windows.

Using automatic hostnames works best for per project installations of Homestead. If you host multiple sites on a single Homestead instance, you may add the "domains" for your web sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead sites into your Homestead virtual machine. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

```
192.168.56.56  homestead.test
```

Make sure the IP address listed is the one set in your `Homestead.yaml` file. Once you have added the domain to your `hosts` file and launched the Vagrant box you will be able to access the site via your web browser:

```
http://homestead.test
```

Configuring Services

Homestead starts several services by default; however, you may customize which services are enabled or disabled during provisioning. For example, you may enable PostgreSQL and disable MySQL by modifying the `services` option within your `Homestead.yaml` file:

```
services:
    - enabled:
        - "postgresql"
    - disabled:
        - "mysql"
```

The specified services will be started or stopped based on their order in the `enabled` and `disabled` directives.

Launching The Vagrant Box

Once you have edited the `Homestead.yaml` to your liking, run the `vagrant up` command from your Homestead directory. Vagrant will boot the virtual machine and automatically configure your shared folders and Nginx sites.

To destroy the machine, you may use the `vagrant destroy` command.

Per Project Installation

Instead of installing Homestead globally and sharing the same Homestead virtual machine across all of your projects, you may instead configure a Homestead instance for each project you manage. Installing Homestead per project may be beneficial if you wish to ship a `Vagrantfile` with your project, allowing others working on the project to `vagrant up` immediately after cloning the project's repository.

You may install Homestead into your project using the Composer package manager:

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

Once Homestead has been installed, invoke Homestead's `make` command to generate the `Vagrantfile` and `Homestead.yaml` file for your project. These files will be placed in the root of your project. The `make` command will automatically configure the `sites` and `folders` directives in the `Homestead.yaml` file:

```
// macOS / Linux...
php vendor/bin/homestead make

// Windows...
vendor\\bin\\homestead make
```

Next, run the `vagrant up` command in your terminal and access your project at `http://homestead.test` in your browser. Remember, you will still need to add an `/etc/hosts` file entry for `homestead.test` or the domain of your choice if you are not using automatic hostname resolution.

Installing Optional Features

Optional software is installed using the `features` option within your `Homestead.yaml` file. Most features can be enabled or disabled with a boolean value, while some features allow multiple configuration options:

```
features:
    - blackfire:
        server_id: "server_id"
        server_token: "server_value"
        client_id: "client_id"
        client_token: "client_value"
    - cassandra: true
    - chronograf: true
    - couchdb: true
    - crystal: true
    - docker: true
    - elasticsearch:
        version: 7.9.0
    - eventstore: true
        version: 21.2.0
    - gearman: true
    - golang: true
    - grafana: true
    - influxdb: true
    - mariadb: true
    - meilisearch: true
    - minio: true
    - mongodb: true
    - neo4j: true
    - ohmyzsh: true
    - openresty: true
    - pm2: true
    - python: true
    - r-base: true
    - rabbitmq: true
    - rvm: true
    - solr: true
    - timescaledb: true
    - trader: true
    - webdriver: true
```

Elasticsearch

You may specify a supported version of Elasticsearch, which must be an exact version number (major.minor.patch). The default installation will create a cluster named 'homestead'. You should never give Elasticsearch more than half of the operating system's memory, so make sure your Homestead virtual machine has at least twice the Elasticsearch allocation.

{tip} Check out the Elasticsearch documentation to learn how to customize your configuration.

MariaDB

Enabling MariaDB will remove MySQL and install MariaDB. MariaDB typically serves as a drop-in replacement for MySQL, so you should still use the `mysql` database driver in your application's database configuration.

MongoDB

The default MongoDB installation will set the database username to `homestead` and the corresponding password to `secret`.

Neo4j

The default Neo4j installation will set the database username to `homestead` and the corresponding password to `secret`. To access the Neo4j browser, visit `http://homestead.test:7474` via your web browser. The ports `7687` (Bolt), `7474` (HTTP), and `7473` (HTTPS) are ready to serve requests from the Neo4j client.

Aliases

You may add Bash aliases to your Homestead virtual machine by modifying the `aliases` file within your Homestead directory:

```
alias c='clear'
alias ..='cd ..'
```

After you have updated the `aliases` file, you should re-provision the Homestead virtual machine using the `vagrant reload --provision` command. This will ensure that your new aliases are available on the machine.

Updating Homestead

Before you begin updating Homestead you should ensure you have removed your current virtual machine by running the following command in your Homestead directory:

```
vagrant destroy
```

Next, you need to update the Homestead source code. If you cloned the repository, you can execute the following commands at the location you originally cloned the repository:

```
git fetch

git pull origin release
```

These commands pull the latest Homestead code from the GitHub repository, fetch the latest tags, and then check out the latest tagged release. You can find the latest stable release version on Homestead's GitHub releases page.

If you have installed Homestead via your project's `composer.json` file, you should ensure your `composer.json` file contains `"laravel/homestead": "^12"` and update your dependencies:

```
composer update
```

Next, you should update the Vagrant box using the `vagrant box update` command:

```
vagrant box update
```

After updating the Vagrant box, you should run the `bash init.sh` command from the Homestead directory in order to update Homestead's additional configuration files. You will be asked whether you wish to overwrite your existing `Homestead.yaml`, `after.sh`, and `aliases` files:

```
// macOS / Linux...
bash init.sh

// Windows...
init.bat
```

Finally, you will need to regenerate your Homestead virtual machine to utilize the latest Vagrant installation:

```
vagrant up
```

Daily Usage

Connecting Via SSH

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

Adding Additional Sites

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

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
    - map: another.test
      to: /home/vagrant/project2/public
```

{note} You should ensure that you have configured a folder mapping for the project's directory before adding the site.

If Vagrant is not automatically managing your "hosts" file, you may need to add the new site to that file as well. On macOS and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`:

```
192.168.56.56  homestead.test
192.168.56.56  another.test
```

Once the site has been added, execute the `vagrant reload --provision` terminal command from your Homestead directory.

Site Types

Homestead supports several "types" of sites which allow you to easily run projects that are not based on Laravel. For example, we may easily add a Statamic application to Homestead using the `statamic` site type:

```
sites:
    - map: statamic.test
      to: /home/vagrant/my-symfony-project/web
      type: "statamic"
```

The available site types are: `apache`, `apigility`, `expressive`, `laravel` (the default), `proxy`, `silverstripe`, `statamic`, `symfony2`, `symfony4`, and `zf`.

Site Parameters

You may add additional Nginx `fastcgi_param` values to your site via the `params` site directive:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      params:
          - key: FOO
            value: BAR
```

Environment Variables

You can define global environment variables by adding them to your `Homestead.yaml` file:

```
variables:
    - key: APP_ENV
      value: local
    - key: FOO
      value: bar
```

After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing

the `vagrant reload --provision` command. This will update the PHP-FPM configuration for all of the installed PHP versions and also update the environment for the `vagrant` user.

Ports

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

- **HTTP:** 8000 → Forwards To 80 - **HTTPS:** 44300 → Forwards To 443

Forwarding Additional Ports

If you wish, you may forward additional ports to the Vagrant box by defining a `ports` configuration entry within your `Homestead.yaml` file. After updating the `Homestead.yaml` file, be sure to re-provision the machine by executing the `vagrant reload --provision` command:

```
ports:
    - send: 50000
      to: 5000
    - send: 7777
      to: 777
      protocol: udp
```

Below is a list of additional Homestead service ports that you may wish to map from your host machine to your Vagrant box:

- **SSH:** 2222 → To 22 - **ngrok UI:** 4040 → To 4040 - **MySQL:** 33060 → To 3306 - **PostgreSQL:** 54320 → To 5432 - **MongoDB:** 27017 → To 27017 - **Mailhog:** 8025 → To 8025 - **Minio:** 9600 → To 9600

PHP Versions

Homestead 6 introduced support for running multiple versions of PHP on the same virtual machine. You may specify which version of PHP to use for a given site within your `Homestead.yaml` file. The available PHP versions are: "5.6", "7.0", "7.1", "7.2", "7.3", "7.4", "8.0" (the default), and "8.1":

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      php: "7.1"
```

Within your Homestead virtual machine, you may use any of the supported PHP versions via the CLI:

```
php5.6 artisan list
php7.0 artisan list
php7.1 artisan list
php7.2 artisan list
php7.3 artisan list
php7.4 artisan list
php8.0 artisan list
php8.1 artisan list
```

You may change the default version of PHP used by the CLI by issuing the following commands from within your Homestead virtual machine:

```
php56
php70
php71
php72
php73
php74
php80
php81
```

Connecting To Databases

A `homestead` database is configured for both MySQL and PostgreSQL out of the box. To connect to your MySQL or PostgreSQL database from your host machine's database client, you should connect to `127.0.0.1` on port `33060` (MySQL) or `54320` (PostgreSQL). The username and password for both databases is `homestead` / `secret`.

{note} You should only use these non-standard ports when connecting to the databases from your host machine. You will use the default 3306 and 5432 ports in your Laravel application's `database` configuration file since Laravel is running *within* the virtual machine.

Database Backups

Homestead can automatically backup your database when your Homestead virtual machine is destroyed. To utilize this feature, you must be using Vagrant 2.1.0 or greater. Or, if you are using an older version of Vagrant, you must install the `vagrant-triggers` plug-in. To enable automatic database backups, add the following line to your `Homestead.yaml` file:

```
backup: true
```

Once configured, Homestead will export your databases to `mysql_backup` and `postgres_backup` directories when the `vagrant destroy` command is executed. These directories can be found in the folder where you installed Homestead or in the root of your project if you are using the per project installation method.

Configuring Cron Schedules

Laravel provides a convenient way to schedule cron jobs by scheduling a single `schedule:run` Artisan command to run every minute. The `schedule:run` command will examine the job schedule defined in your `App\Console\Kernel` class to determine which scheduled tasks to run.

If you would like the `schedule:run` command to be run for a Homestead site, you may set the `schedule` option to `true` when defining the site:

```
sites:
    - map: homestead.test
      to: /home/vagrant/project1/public
      schedule: true
```

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

Configuring MailHog

MailHog allows you to intercept your outgoing email and examine it without actually sending the mail to its recipients. To get started, update your application's `.env` file to use the following mail settings:

```
MAIL_MAILER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Once MailHog has been configured, you may access the MailHog dashboard at `http://localhost:8025`.

Configuring Minio

Minio is an open source object storage server with an Amazon S3 compatible API. To install Minio, update your `Homestead.yaml` file with the following configuration option in the features section:

```
minio: true
```

By default, Minio is available on port 9600. You may access the Minio control panel by visiting `http://localhost:9600`. The default access key is `homestead`, while the default secret key is `secretkey`. When accessing Minio, you should always use region `us-east-1`.

In order to use Minio, you will need to adjust the S3 disk configuration in your application's `config/filesystems.php` configuration file. You will need to add the `use_path_style_endpoint` option to the disk configuration as well as change the `url` key to `endpoint`:

```
's3' => [
    'driver' => 's3',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION'),
    'bucket' => env('AWS_BUCKET'),
    'endpoint' => env('AWS_URL'),
    'use_path_style_endpoint' => true,
]
```

Finally, ensure your `.env` file has the following options:

```
AWS_ACCESS_KEY_ID=homestead
AWS_SECRET_ACCESS_KEY=secretkey
AWS_DEFAULT_REGION=us-east-1
AWS_URL=http://localhost:9600
```

To provision Minio powered "S3" buckets, add a `buckets` directive to your `Homestead.yaml` file. After defining your buckets, you should execute the `vagrant reload --provision` command in your terminal:

```
buckets:
    - name: your-bucket
      policy: public
    - name: your-private-bucket
      policy: none
```

Supported `policy` values include: `none`, `download`, `upload`, and `public`.

Laravel Dusk

In order to run Laravel Dusk tests within Homestead, you should enable the `webdriver` feature in your Homestead configuration:

```
features:
    - webdriver: true
```

After enabling the `webdriver` feature, you should execute the `vagrant reload --provision` command in your terminal.

Sharing Your Environment

Sometimes you may wish to share what you're currently working on with coworkers or a client. Vagrant has built-in support for this via the `vagrant share` command; however, this will not work if you have multiple sites configured in your `Homestead.yaml` file.

To solve this problem, Homestead includes its own `share` command. To get started, SSH into your Homestead virtual machine via `vagrant ssh` and execute the `share homestead.test` command. This command will share the `homestead.test` site from your `Homestead.yaml` configuration file. You may substitute any of your other configured sites for `homestead.test`:

```
share homestead.test
```

After running the command, you will see an Ngrok screen appear which contains the activity log and the publicly accessible URLs for the shared site. If you would like to specify a custom region, subdomain, or other Ngrok runtime option, you may add them to your `share` command:

```
share homestead.test -region=eu -subdomain=laravel
```

{note} Remember, Vagrant is inherently insecure and you are exposing your virtual machine to the Internet when running the `share` command.

Debugging & Profiling

Debugging Web Requests With Xdebug

Homestead includes support for step debugging using Xdebug. For example, you can access a page in your browser and PHP will connect to your IDE to allow inspection and modification of the running code.

By default, Xdebug is already running and ready to accept connections. If you need to enable Xdebug on the CLI, execute the `sudo phpenmod xdebug` command within your Homestead virtual machine. Next, follow your IDE's instructions to enable debugging. Finally, configure your browser to trigger Xdebug with an extension or bookmarklet.

{note} Xdebug causes PHP to run significantly slower. To disable Xdebug, run `sudo phpdismod xdebug` within your Homestead virtual machine and restart the FPM service.

Autostarting Xdebug

When debugging functional tests that make requests to the web server, it is easier to autostart debugging rather than modifying tests to pass through a custom header or cookie to trigger debugging. To force Xdebug to start automatically, modify the `/etc/php/7.x/fpm/conf.d/20-xdebug.ini` file inside your Homestead virtual machine and add the following configuration:

```
; If Homestead.yaml contains a different subnet for the IP address, this
address may be different...
xdebug.remote_host = 192.168.10.1
xdebug.remote_autostart = 1
```

Debugging CLI Applications

To debug a PHP CLI application, use the `xphp` shell alias inside your Homestead virtual machine:

```
xphp /path/to/script
```

Profiling Applications with Blackfire

Blackfire is a service for profiling web requests and CLI applications. It offers an interactive user interface which displays profile data in call-graphs and timelines. It is built for use in development, staging, and production, with no overhead for end users. In addition, Blackfire provides performance, quality, and security checks on code and `php.ini` configuration settings.

The Blackfire Player is an open-source Web Crawling, Web Testing, and Web Scraping application which can work jointly with Blackfire in order to script profiling scenarios.

To enable Blackfire, use the "features" setting in your Homestead configuration file:

```
features:
    - blackfire:
        server_id: "server_id"
        server_token: "server_value"
        client_id: "client_id"
        client_token: "client_value"
```

Blackfire server credentials and client credentials require a Blackfire account. Blackfire offers various options to profile an application, including a CLI tool and browser extension. Please review the Blackfire documentation for more details.

Network Interfaces

The `networks` property of the `Homestead.yaml` file configures network interfaces for your Homestead virtual machine. You may configure as many interfaces as necessary:

```
networks:
    - type: "private_network"
      ip: "192.168.10.20"
```

To enable a bridged interface, configure a `bridge` setting for the network and change the network type to `public_network`:

```
networks:
    - type: "public_network"
      ip: "192.168.10.20"
      bridge: "en1: Wi-Fi (AirPort)"
```

To enable DHCP, just remove the `ip` option from your configuration:

```
networks:
    - type: "public_network"
      bridge: "en1: Wi-Fi (AirPort)"
```

Extending Homestead

You may extend Homestead using the `after.sh` script in the root of your Homestead directory. Within this file, you may add any shell commands that are necessary to properly configure and customize your virtual machine.

When customizing Homestead, Ubuntu may ask you if you would like to keep a package's original configuration or overwrite it with a new configuration file. To avoid this, you should use the following command when installing packages in order to avoid overwriting any configuration previously written by Homestead:

```
sudo apt-get -y \
    -o Dpkg::Options::="--force-confdef" \
    -o Dpkg::Options::="--force-confold" \
    install package-name
```

User Customizations

When using Homestead with your team, you may want to tweak Homestead to better fit your personal development style. To accomplish this, you may create a `user-customizations.sh` file in the root of your Homestead directory (the same directory containing your `Homestead.yaml` file). Within this file, you may make any customization you would like; however, the `user-customizations.sh` should not be version controlled.

Provider Specific Settings

VirtualBox

`natdnshostresolver`

By default, Homestead configures the `natdnshostresolver` setting to `on`. This allows Homestead to use your host operating system's DNS settings. If you would like to override this behavior, add the following configuration options to your `Homestead.yaml` file:

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

Symbolic Links On Windows

If symbolic links are not working properly on your Windows machine, you may need to add the following block to your `Vagrantfile`:

```
config.vm.provider "virtualbox" do |v|
    v.customize ["setextradata", :id,
"VBoxInternal2/SharedFoldersEnableSymlinksCreate/v-root", "1"]
end
```

Laravel Horizon

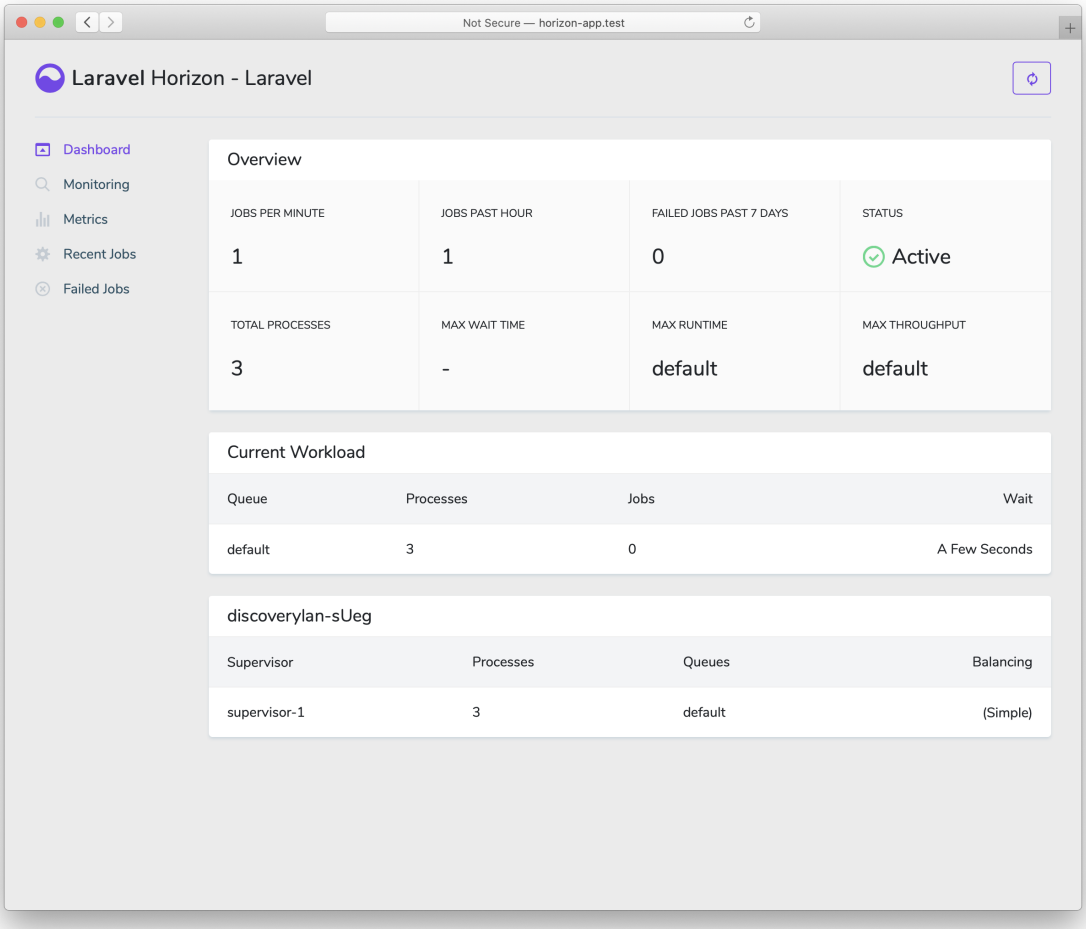
- Notifications
- Metrics
- Deleting Failed Jobs
- Clearing Jobs From Queues

Introduction

{tip} Before digging into Laravel Horizon, you should familiarize yourself with Laravel's base queue services. Horizon augments Laravel's queue with additional features that may be confusing if you are not already familiar with the basic queue features offered by Laravel.

Laravel Horizon provides a beautiful dashboard and code-driven configuration for your Laravel powered Redis queues. Horizon allows you to easily monitor key metrics of your queue system such as job throughput, runtime, and job failures.

When using Horizon, all of your queue worker configuration is stored in a single, simple configuration file. By defining your application's worker configuration in a version controlled file, you may easily scale or modify your application's queue workers when deploying your application.



Installation

{note} Laravel Horizon requires that you use Redis to power your queue. Therefore, you should ensure that your queue connection is set to `redis` in your application's `config/queue.php` configuration file.

You may install Horizon into your project using the Composer package manager:

```
composer require laravel/horizon
```

After installing Horizon, publish its assets using the `horizon:install` Artisan command:

```
php artisan horizon:install
```

Configuration

After publishing Horizon's assets, its primary configuration file will be located at `config/horizon.php`. This configuration file allows you to configure the queue worker options for your application. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.

Environments

After installation, the primary Horizon configuration option that you should familiarize yourself with is the `environments` configuration option. This configuration option is an array of environments that your application runs on and defines the worker process options for each environment. By default, this entry contains a `production` and `local` environment. However, you are free to add more environments as needed:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
        ],
    ],

    'local' => [
        'supervisor-1' => [
            'maxProcesses' => 3,
        ],
    ],
],
```

When you start Horizon, it will use the worker process configuration options for the environment that your application is running on. Typically, the environment is determined by the value of the `APP_ENV` environment variable. For example, the default `local` Horizon environment is configured to start three worker processes and automatically balance the number of worker processes assigned to each queue. The default `production` environment is configured to start a maximum of 10 worker processes and automatically balance the number of worker processes assigned to each queue.

{note} You should ensure that the `environments` portion of your `horizon` configuration file contains an entry for each environment on which you plan to run Horizon.

Supervisors

As you can see in Horizon's default configuration file. Each environment can contain one or more "supervisors". By default, the configuration file defines this supervisor as `supervisor-1`; however, you are free to name your supervisors whatever you want. Each supervisor is essentially responsible for "supervising" a group of worker processes and takes care of balancing worker processes across queues.

You may add additional supervisors to a given environment if you would like to define a new group of worker processes that should run in that environment. You may choose to do this if you would like to define a different balancing strategy or worker process count for a given queue used by your application.

Default Values

Within Horizon's default configuration file, you will notice a `defaults` configuration option. This configuration option specifies the default values for your application's supervisors. The supervisor's default configuration values will be merged into the supervisor's configuration for each environment, allowing you to avoid unnecessary repetition when defining your supervisors.

Balancing Strategies

Unlike Laravel's default queue system, Horizon allows you to choose from three worker balancing strategies: `simple`, `auto`, and `false`. The `simple` strategy, which is the configuration file's default, splits incoming jobs evenly between worker processes:

```
'balance' => 'simple',
```

The `auto` strategy adjusts the number of worker processes per queue based on the current workload of the queue. For example, if your `notifications` queue has 1,000 pending jobs while your `render` queue is empty, Horizon will allocate more workers to your `notifications` queue until the queue is empty.

When using the `auto` strategy, you may define the `minProcesses` and `maxProcesses` configuration options to control the minimum and the maximum number of worker processes Horizon should scale up and down to:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default'],
            'balance' => 'auto',
            'minProcesses' => 1,
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
            'tries' => 3,
        ],
    ],
],
```

The `balanceMaxShift` and `balanceCooldown` configuration values to determine how quickly Horizon will scale to meet worker demand. In the example above, a maximum of one new process will be created or destroyed every three seconds. You are free to tweak these values as necessary based on your application's needs.

When the `balance` option is set to `false`, the default Laravel behavior will be used, which processes queues in the order they are listed in your configuration.

Dashboard Authorization

Horizon exposes a dashboard at the `/horizon` URI. By default, you will only be able to access this dashboard in the `local` environment. However, within your `app/Providers/HorizonServiceProvider.php` file, there is an authorization gate definition. This authorization gate controls access to Horizon in **non-local** environments. You are free to modify this gate as needed to restrict access to your Horizon installation:

```
/**
 * Register the Horizon gate.
 *
 * This gate determines who can access Horizon in non-local
environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewHorizon', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

Alternative Authentication Strategies

Remember that Laravel automatically injects the authenticated user into the gate closure. If your application is providing Horizon security via another method, such as IP restrictions, then your Horizon users may not need to "login". Therefore, you will need to change `function ($user)` closure signature above to `function ($user = null)` in order to force Laravel to not require authentication.

Upgrading Horizon

When upgrading to a new major version of Horizon, it's important that you carefully review the upgrade guide. In addition, when upgrading to any new Horizon version, you should re-publish Horizon's assets:

```
php artisan horizon:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `horizon:publish` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan horizon:publish --ansi"
        ]
    }
}
```

Running Horizon

Once you have configured your supervisors and workers in your application's `config/horizon.php` configuration file, you may start Horizon using the `horizon` Artisan command. This single command will start all of the configured worker processes for the current environment:

```
php artisan horizon
```

You may pause the Horizon process and instruct it to continue processing jobs using the `horizon:pause` and `horizon:continue` Artisan commands:

```
php artisan horizon:pause

php artisan horizon:continue
```

You may also pause and continue specific Horizon supervisors using the `horizon:pause-supervisor` and `horizon:continue-supervisor` Artisan commands:

```
php artisan horizon:pause-supervisor supervisor-1

php artisan horizon:continue-supervisor supervisor-1
```

You may check the current status of the Horizon process using the `horizon:status` Artisan command:

```
php artisan horizon:status
```

You may gracefully terminate the Horizon process using the `horizon:terminate` Artisan command. Any jobs that are currently being processed by will be completed and then Horizon will stop executing:

```
php artisan horizon:terminate
```

Deploying Horizon

When you're ready to deploy Horizon to your application's actual server, you should configure a process monitor to monitor the `php artisan horizon` command and restart it if it exits unexpectedly. Don't worry, we'll discuss how to install a process monitor below.

During your application's deployment process, you should instruct the Horizon process to terminate so that it will be restarted by your process monitor and receive your code changes:

```
php artisan horizon:terminate
```

Installing Supervisor

Supervisor is a process monitor for the Linux operating system and will automatically restart your `horizon` process if it stops executing. To install Supervisor on Ubuntu, you may use the following command. If you are not using Ubuntu, you can likely install Supervisor using your operating system's package manager:

```
sudo apt-get install supervisor
```

{tip} If configuring Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your Laravel projects.

Supervisor Configuration

Supervisor configuration files are typically stored within your server's `/etc/supervisor/conf.d` directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a `horizon.conf` file that starts and monitors a `horizon` process:

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/example.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/example.com/horizon.log
stopwaitsecs=3600
```

{note} You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the monitored processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start horizon
```

{tip} For more information on running Supervisor, consult the Supervisor documentation.

Tags

Horizon allows you to assign “tags” to jobs, including mailables, broadcast events, notifications, and queued event listeners. In fact, Horizon will intelligently and automatically tag most jobs depending on the Eloquent models that are attached to the job. For example, take a look at the following job:

```php
<?php

namespace App\Jobs;

use App\Models\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The video instance.
     *
     * @var \App\Models\Video
     */
    public $video;

    /**
     * Create a new job instance.
     *
     * @param  \App\Models\Video  $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

If this job is queued with an `App\Models\Video` instance that has an `id` attribute of `1`, it will automatically receive the tag `App\Models\Video:1`. This is because Horizon will search the job's properties for any Eloquent models. If Eloquent models are found, Horizon will intelligently tag the job using the model's class name and primary key:

```
use App\Jobs\RenderVideo;
use App\Models\Video;

$video = Video::find(1);

RenderVideo::dispatch($video);
```

Manually Tagging Jobs

If you would like to manually define the tags for one of your queueable objects, you may define a `tags` method on the class:

```
class RenderVideo implements ShouldQueue
{
    /**
     * Get the tags that should be assigned to the job.
     *
     * @return array
     */
    public function tags()
    {
        return ['render', 'video:'.$this->video->id];
    }
}
```

Notifications

{note} When configuring Horizon to send Slack or SMS notifications, you should review the prerequisites for the relevant notification channel.

If you would like to be notified when one of your queues has a long wait time, you may use the `Horizon::routeMailNotificationsTo`, `Horizon::routeSlackNotificationsTo`, and `Horizon::routeSmsNotificationsTo` methods. You may call these methods from the `boot` method of your application's `App\Providers\HorizonServiceProvider`:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    parent::boot();

    Horizon::routeSmsNotificationsTo('15556667777');
    Horizon::routeMailNotificationsTo('example@example.com');
    Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
}
```

Configuring Notification Wait Time Thresholds

You may configure how many seconds are considered a "long wait" within your application's `config/horizon.php` configuration file. The `waits` configuration option within this file allows you to control the long wait threshold for each connection / queue combination:

```
'waits' => [
    'redis:default' => 60,
    'redis:critical,high' => 90,
],
```

Metrics

Horizon includes a metrics dashboard which provides information regarding your job and queue wait times and throughput. In order to populate this dashboard, you should configure Horizon's `snapshot` Artisan command to run every five minutes via your application's scheduler:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

Deleting Failed Jobs

If you would like to delete a failed job, you may use the `horizon:forget` command. The `horizon:forget` command accepts the ID or UUID of the failed job as its only argument:

```
php artisan horizon:forget 5
```

Clearing Jobs From Queues

If you would like to delete all jobs from your application's default queue, you may do so using the `horizon:clear` Artisan command:

```
php artisan horizon:clear
```

You may provide the `queue` option to delete jobs from a specific queue:

```
php artisan horizon:clear --queue=emails
```

HTTP Tests

Introduction

Laravel provides a very fluent API for making HTTP requests to your application and examining the responses. For example, take a look at the feature test defined below:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_a_basic_request()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

The `get` method makes a `GET` request into the application, while the `assertStatus` method asserts that the returned response should have the given HTTP status code. In addition to this simple assertion, Laravel also contains a variety of assertions for inspecting the response headers, content, JSON structure, and more.

Making Requests

To make a request to your application, you may invoke the `get`, `post`, `put`, `patch`, or `delete` methods within your test. These methods do not actually issue a "real" HTTP request to your application. Instead, the entire network request is simulated internally.

Instead of returning an `Illuminate\Http\Response` instance, test request methods return an instance of `Illuminate\Testing\TestResponse`, which provides a variety of helpful assertions that allow you to inspect your application's responses:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_a_basic_request()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

In general, each of your tests should only make one request to your application. Unexpected behavior may occur if multiple requests are executed within a single test method.

{tip} For convenience, the CSRF middleware is automatically disabled when running tests.

Customizing Request Headers

You may use the `withHeaders` method to customize the request's headers before it is sent to the application. This method allows you to add any custom headers you would like to the request:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_interacting_with_headers()
    {
        $response = $this->withHeaders([
            'X-Header' => 'Value',
        ])->post('/user', ['name' => 'Sally']);

        $response->assertStatus(201);
    }
}
```

Cookies

You may use the `withCookie` or `withCookies` methods to set cookie values before making a request. The `withCookie` method accepts a cookie name and value as its two arguments, while the `withCookies` method accepts an array of name / value pairs:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_cookies()
    {
        $response = $this->withCookie('color', 'blue')->get('/');

        $response = $this->withCookies([
            'color' => 'blue',
            'name' => 'Taylor',
        ])->get('/');
    }
}
```

Session / Authentication

Laravel provides several helpers for interacting with the session during HTTP testing. First, you may set the session data to a given array using the `withSession` method. This is useful for loading the session with data before issuing a request to your application:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_the_session()
    {
        $response = $this->withSession(['banned' => false])->get('/');
    }
}
```

Laravel's session is typically used to maintain state for the currently authenticated user.

Therefore, the `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a model factory to generate and authenticate a user:

```
<?php

namespace Tests\Feature;

use App\Models\User;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_an_action_that_requires_authentication()
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
                         ->withSession(['banned' => false])
                         ->get('/');
    }
}
```

You may also specify which guard should be used to authenticate the given user by passing the guard name as the second argument to the `actingAs` method:

```
$this->actingAs($user, 'web')
```

Debugging Responses

After making a test request to your application, the `dump`, `dumpHeaders`, and `dumpSession` methods may be used to examine and debug the response contents:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_basic_test()
    {
        $response = $this->get('/');

        $response->dumpHeaders();

        $response->dumpSession();

        $response->dump();
    }
}
```

Alternatively, you may use the `dd`, `ddHeaders`, and `ddSession` methods to dump information about the response and then stop execution:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_basic_test()
    {
        $response = $this->get('/');

        $response->ddHeaders();

        $response->ddSession();

        $response->dd();
    }
}
```

Exception Handling

Sometimes you may want to test that your application is throwing a specific exception. To ensure that the exception does not get caught by Laravel's exception handler and returned as an HTTP response, you may invoke the `withoutExceptionHandling` method before making your request:

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

In addition, if you would like to ensure that your application is not utilizing features that have been deprecated by the PHP language or the libraries your application is using, you may invoke the `withoutDeprecationHandling` method before making your request. When deprecation handling is disabled, deprecation warnings will be converted to exceptions, thus causing your test to fail:

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

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `json`, `getJson`, `postJson`, `putJson`, `patchJson`, `deleteJson`, and `optionsJson` methods may be used to issue JSON requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/api/user` and assert that the expected JSON data was returned:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_making_an_api_request()
    {
        $response = $this->postJson('/api/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

In addition, JSON response data may be accessed as array variables on the response, making it convenient for you to inspect the individual values returned within a JSON response:

```
$this->assertTrue($response['created']);
```

{tip} The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

Asserting Exact JSON Matches

As previously mentioned, the `assertJson` method may be used to assert that a fragment of JSON exists within the JSON response. If you would like to verify that a given array **exactly matches** the JSON returned by your application, you should use the `assertExactJson` method:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_asserting_an_exact_json_match()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

Asserting On JSON Paths

If you would like to verify that the JSON response contains the given data at a specified path, you should use the `assertJsonPath` method:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function test_asserting_a_json_paths_value()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertJsonPath('team.owner.name', 'Darian');
    }
}
```

Fluent JSON Testing

Laravel also offers a beautiful way to fluently test your application's JSON responses. To get started, pass a closure to the `assertJson` method. This closure will be invoked with an instance of `Illuminate\Testing\Fluent\AssertableJson` which can be used to make assertions against the JSON that was returned by your application. The `where` method may be used to make assertions against a particular attribute of the JSON, while the `missing` method may be used to assert that a particular attribute is missing from the JSON:

```
use Illuminate\Testing\Fluent\AssertableJson;

/**
 * A basic functional test example.
 *
 * @return void
 */
public function test_fluent_json()
{
    $response = $this->json('GET', '/users/1');

    $response
        ->assertJson(fn (AssertableJson $json) =>
            $json->where('id', 1)
                 ->where('name', 'Victoria Faith')
                 ->missing('password')
                 ->etc()
        );
}
```

Understanding The `etc` Method

In the example above, you may have noticed we invoked the `etc` method at the end of our assertion chain. This method informs Laravel that there may be other attributes present on the JSON object. If the `etc` method is not used, the test will fail if other attributes that you did not make assertions against exist on the JSON object.

The intention behind this behavior is to protect you from unintentionally exposing sensitive information in your JSON responses by forcing you to either explicitly make an assertion against the attribute or explicitly allow additional attributes via the `etc` method.

Asserting Attribute Presence / Absence

To assert that an attribute is present or absent, you may use the `has` and `missing` methods:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('data')
         ->missing('message')
);
```

In addition, the `hasAll` and `missingAll` methods allow asserting the presence or absence

of multiple attributes simultaneously:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->hasAll('status', 'data')
         ->missingAll('message', 'code')
);
```

You may use the `hasAny` method to determine if at least one of a given list of attributes is present:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('status')
         ->hasAny('data', 'message', 'code')
);
```

Asserting Against JSON Collections

Often, your route will return a JSON response that contains multiple items, such as multiple users:

```
Route::get('/users', function () {
    return User::all();
});
```

In these situations, we may use the fluent JSON object's `has` method to make assertions against the users included in the response. For example, let's assert that the JSON response contains three users. Next, we'll make some assertions about the first user in the collection using the `first` method. The `first` method accepts a closure which receives another assertable JSON string that we can use to make assertions about the first object in the JSON collection:

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has(3)
             ->first(fn ($json) =>
                $json->where('id', 1)
                     ->where('name', 'Victoria Faith')
                     ->missing('password')
                     ->etc()
             )
    );
```

Scoping JSON Collection Assertions

Sometimes, your application's routes will return JSON collections that are assigned named keys:

```
Route::get('/users', function () {
    return [
        'meta' => [...],
        'users' => User::all(),
    ];
})
```

When testing these routes, you may use the has method to assert against the number of items in the collection. In addition, you may use the has method to scope a chain of assertions:

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has('meta')
             ->has('users', 3)
             ->has('users.0', fn ($json) =>
                $json->where('id', 1)
                     ->where('name', 'Victoria Faith')
                     ->missing('password')
                     ->etc()
             )
    );
```

However, instead of making two separate calls to the `has` method to assert against the `users` collection, you may make a single call which provides a closure as its third parameter. When doing so, the closure will automatically be invoked and scoped to the first item in the collection:

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has('meta')
             ->has('users', 3, fn ($json) =>
                $json->where('id', 1)
                     ->where('name', 'Victoria Faith')
                     ->missing('password')
                     ->etc()
             )
    );
```

Asserting JSON Types

You may only want to assert that the properties in the JSON response are of a certain type. The `Illuminate\Testing\Fluent\AssertableJson` class provides the `whereType` and `whereAllType` methods for doing just that:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->whereType('id', 'integer')
         ->whereAllType([
            'users.0.name' => 'string',
            'meta' => 'array'
        ])
);
```

You may specify multiple types using the `|` character, or passing an array of types as the second parameter to the `whereType` method. The assertion will be successful if the response value is any of the listed types:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->whereType('name', 'string|null')
         ->whereType('id', ['string', 'integer'])
);
```

The `whereType` and `whereAllType` methods recognize the following types: `string`, `integer`, `double`, `boolean`, `array`, and `null`.

Testing File Uploads

The `Illuminate\Http\UploadedFile` class provides a `fake` method which may be used to generate dummy files or images for testing. This, combined with the `Storage` facade's `fake` method, greatly simplifies the testing of file uploads. For example, you may combine these two features to easily test an avatar upload form:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_avatars_can_be_uploaded()
    {
        Storage::fake('avatars');

        $file = UploadedFile::fake()->image('avatar.jpg');

        $response = $this->post('/avatar', [
            'avatar' => $file,
        ]);

        Storage::disk('avatars')->assertExists($file->hashName());
    }
}
```

If you would like to assert that a given file does not exist, you may use the `assertMissing` method provided by the `Storage` facade:

```
Storage::fake('avatars');

// ...

Storage::disk('avatars')->assertMissing('missing.jpg');
```

Fake File Customization

When creating files using the `fake` method provided by the `UploadedFile` class, you may specify the width, height, and size of the image (in kilobytes) in order to better test your application's validation rules:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

In addition to creating images, you may create files of any other type using the `create` method:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

If needed, you may pass a `$mimeType` argument to the method to explicitly define the MIME type that should be returned by the file:

```
UploadedFile::fake()->create(
    'document.pdf', $sizeInKilobytes, 'application/pdf'
);
```

Testing Views

Laravel also allows you to render a view without making a simulated HTTP request to the application. To accomplish this, you may call the `view` method within your test. The `view` method accepts the view name and an optional array of data. The method returns an instance of `Illuminate\Testing\TestView`, which offers several methods to conveniently make assertions about the view's contents:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_a_welcome_view_can_be_rendered()
    {
        $view = $this->view('welcome', ['name' => 'Taylor']);

        $view->assertSee('Taylor');
    }
}
```

The `TestView` class provides the following assertion methods: `assertSee`, `assertSeeInOrder`, `assertSeeText`, `assertSeeTextInOrder`, `assertDontSee`, and `assertDontSeeText`.

If needed, you may get the raw, rendered view contents by casting the `TestView` instance to a string:

```
$contents = (string) $this->view('welcome');
```

Sharing Errors

Some views may depend on errors shared in the global error bag provided by Laravel. To hydrate the error bag with error messages, you may use the `withViewErrors` method:

```
$view = $this->withViewErrors([
    'name' => ['Please provide a valid name.']
])->view('form');

$view->assertSee('Please provide a valid name.');
```

Rendering Blade & Components

If necessary, you may use the `blade` method to evaluate and render a raw Blade string. Like the `view` method, the `blade` method returns an instance of `Illuminate\Testing\TestView`:

```
$view = $this->blade(
    '<x-component :name="$name" />',
    ['name' => 'Taylor']
);

$view->assertSee('Taylor');
```

You may use the `component` method to evaluate and render a Blade component. Like the `view` method, the `component` method returns an instance of `Illuminate\Testing\TestView`:

```
$view = $this->component(Profile::class, ['name' => 'Taylor']);

$view->assertSee('Taylor');
```

Available Assertions

Response Assertions

Laravel's `Illuminate\Testing\TestResponse` class provides a variety of custom assertion methods that you may utilize when testing your application. These assertions may be accessed on the response that is returned by the `json`, `get`, `post`, `put`, and `delete` test methods:

assertCookie assertCookieExpired assertCookieNotExpired assertCookieMissing assertCreated assertDontSee assertDontSeeText assertDownload assertExactJson assertForbidden assertHeader assertHeaderMissing assertJson assertJsonCount assertJsonFragment assertJsonMissing assertJsonMissingExact assertJsonMissingValidationErrors assertJsonPath assertJsonStructure assertJsonValidationErrors assertLocation assertNoContent assertNotFound assertOk assertPlainCookie assertRedirect assertRedirectToSignedRoute assertSee assertSeeInOrder assertSeeText assertSeeTextInOrder assertSessionHas assertSessionHasInput assertSessionHasAll assertSessionHasErrors assertSessionHasErrorsIn assertSessionHasNoErrors assertSessionDoesntHaveErrors assertSessionMissing assertStatus assertSuccessful assertUnauthorized assertUnprocessable assertValid assertInvalid assertViewHas assertViewHasAll assertViewIs assertViewMissing

assertCookie

Assert that the response contains the given cookie:

```
$response->assertCookie($cookieName, $value = null);
```

assertCookieExpired

Assert that the response contains the given cookie and it is expired:

```
$response->assertCookieExpired($cookieName);
```

assertCookieNotExpired

Assert that the response contains the given cookie and it is not expired:

```
$response->assertCookieNotExpired($cookieName);
```

assertCookieMissing

Assert that the response does not contains the given cookie:

```
$response->assertCookieMissing($cookieName);
```

assertCreated

Assert that the response has a 201 HTTP status code:

```
$response->assertCreated();
```

assertDontSee

Assert that the given string is not contained within the response returned by the application. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertDontSee($value, $escaped = true);
```

assertDontSeeText

Assert that the given string is not contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. This method will pass the response content to the `strip_tags` PHP function before making the assertion:

```
$response->assertDontSeeText($value, $escaped = true);
```

assertDownload

Assert that the response is a "download". Typically, this means the invoked route that returned the response returned a `Response::download` response, `BinaryFileResponse`, or `Storage::download` response:

```
$response->assertDownload();
```

If you wish, you may assert that the downloadable file was assigned a given file name:

```
$response->assertDownload('image.jpg');
```

assertExactJson

Assert that the response contains an exact match of the given JSON data:

```
$response->assertExactJson(array $data);
```

assertForbidden

Assert that the response has a forbidden (403) HTTP status code:

```
$response->assertForbidden();
```

assertHeader

Assert that the given header and value is present on the response:

```
$response->assertHeader($headerName, $value = null);
```

assertHeaderMissing

Assert that the given header is not present on the response:

```
$response->assertHeaderMissing($headerName);
```

assertJson

Assert that the response contains the given JSON data:

```
$response->assertJson(array $data, $strict = false);
```

The `assertJson` method converts the response to an array and utilizes `PHPUnit::assertArraySubset` to verify that the given array exists within the JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

assertJsonCount

Assert that the response JSON has an array with the expected number of items at the given key:

```
$response->assertJsonCount($count, $key = null);
```

assertJsonFragment

Assert that the response contains the given JSON data anywhere in the response:

```
Route::get('/users', function () {
    return [
        'users' => [
            [
                'name' => 'Taylor Otwell',
            ],
        ],
    ];
});

$response->assertJsonFragment(['name' => 'Taylor Otwell']);
```

assertJsonMissing

Assert that the response does not contain the given JSON data:

```
$response->assertJsonMissing(array $data);
```

assertJsonMissingExact

Assert that the response does not contain the exact JSON data:

```
$response->assertJsonMissingExact(array $data);
```

assertJsonMissingValidationErrors

Assert that the response has no JSON validation errors for the given keys:

```
$response->assertJsonMissingValidationErrors($keys);
```

{tip} The more generic assertValid method may be used to assert that a response does not have validation errors that were returned as JSON **and** that no errors were flashed to session storage.

assertJsonPath

Assert that the response contains the given data at the specified path:

```
$response->assertJsonPath($path, $expectedValue);
```

For example, if the JSON response returned by your application contains the following data:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the `name` property of the `user` object matches a given value like so:

```
$response->assertJsonPath('user.name', 'Steve Schoger');
```

assertJsonStructure

Assert that the response has a given JSON structure:

```
$response->assertJsonStructure(array $structure);
```

For example, if the JSON response returned by your application contains the following data:

```
{
    "user": {
        "name": "Steve Schoger"
    }
}
```

You may assert that the JSON structure matches your expectations like so:

```
$response->assertJsonStructure([
    'user' => [
        'name',
    ]
]);
```

Sometimes, JSON responses returned by your application may contain arrays of objects:

```
{
    "user": [
        {
            "name": "Steve Schoger",
            "age": 55,
            "location": "Earth"
        },
        {
            "name": "Mary Schoger",
            "age": 60,
            "location": "Earth"
        }
    ]
}
```

In this situation, you may use the * character to assert against the structure of all of the objects in the array:

```
$response->assertJsonStructure([
    'user' => [
        '*' => [
             'name',
             'age',
             'location'
        ]
    ]
]);
```

assertJsonValidationErrors

Assert that the response has the given JSON validation errors for the given keys. This

method should be used when asserting against responses where the validation errors are returned as a JSON structure instead of being flashed to the session:

```
$response->assertJsonValidationErrors(array $data);
```

{tip} The more generic assertInvalid method may be used to assert that a response has validation errors returned as JSON **or** that errors were flashed to session storage.

assertLocation

Assert that the response has the given URI value in the `Location` header:

```
$response->assertLocation($uri);
```

assertNoContent

Assert that the response has the given HTTP status code and no content:

```
$response->assertNoContent($status = 204);
```

assertNotFound

Assert that the response has a not found (404) HTTP status code:

```
$response->assertNotFound();
```

assertOk

Assert that the response has a 200 HTTP status code:

```
$response->assertOk();
```

assertPlainCookie

Assert that the response contains the given unencrypted cookie:

```
$response->assertPlainCookie($cookieName, $value = null);
```

assertRedirect

Assert that the response is a redirect to the given URI:

```
$response->assertRedirect($uri);
```

assertRedirectToSignedRoute

Assert that the response is a redirect to the given signed route:

```
$response->assertRedirectToSignedRoute($name = null, $parameters = []);
```

assertSee

Assert that the given string is contained within the response. This assertion will automatically escape the given string unless you pass a second argument of `false`:

```
$response->assertSee($value, $escaped = true);
```

assertSeeInOrder

Assert that the given strings are contained in order within the response. This assertion will automatically escape the given strings unless you pass a second argument of `false`:

```
$response->assertSeeInOrder(array $values, $escaped = true);
```

assertSeeText

Assert that the given string is contained within the response text. This assertion will automatically escape the given string unless you pass a second argument of `false`. The response content will be passed to the `strip_tags` PHP function before the assertion is made:

```
$response->assertSeeText($value, $escaped = true);
```

assertSeeTextInOrder

Assert that the given strings are contained in order within the response text. This assertion will automatically escape the given strings unless you pass a second argument of `false`. The response content will be passed to the `strip_tags` PHP function before the assertion is made:

```
$response->assertSeeTextInOrder(array $values, $escaped = true);
```

assertSessionHas

Assert that the session contains the given piece of data:

```
$response->assertSessionHas($key, $value = null);
```

If needed, a closure can be provided as the second argument to the `assertSessionHas` method. The assertion will pass if the closure returns `true`:

```
$response->assertSessionHas($key, function ($value) {
    return $value->name === 'Taylor Otwell';
});
```

assertSessionHasInput

Assert that the session has a given value in the flashed input array:

```
$response->assertSessionHasInput($key, $value = null);
```

If needed, a closure can be provided as the second argument to the `assertSessionHasInput` method. The assertion will pass if the closure returns `true`:

```
$response->assertSessionHasInput($key, function ($value) {
    return Crypt::decryptString($value) === 'secret';
});
```

assertSessionHasAll

Assert that the session contains a given array of key / value pairs:

```
$response->assertSessionHasAll(array $data);
```

For example, if your application's session contains `name` and `status` keys, you may assert that both exist and have the specified values like so:

```
$response->assertSessionHasAll([
    'name' => 'Taylor Otwell',
    'status' => 'active',
]);
```

assertSessionHasErrors

Assert that the session contains an error for the given `$keys`. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key). This method should be used when testing routes that flash validation errors to the session instead of returning them as a JSON structure:

```
$response->assertSessionHasErrors(
    array $keys, $format = null, $errorBag = 'default'
);
```

For example, to assert that the `name` and `email` fields have validation error messages that

were flashed to the session, you may invoke the `assertSessionHasErrors` method like so:

```
$response->assertSessionHasErrors(['name', 'email']);
```

Or, you may assert that a given field has a particular validation error message:

```
$response->assertSessionHasErrors([
    'name' => 'The given name was invalid.'
]);
```

assertSessionHasErrorsIn

Assert that the session contains an error for the given `$keys` within a specific error bag. If `$keys` is an associative array, assert that the session contains a specific error message (value) for each field (key), within the error bag:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format =
null);
```

assertSessionHasNoErrors

Assert that the session has no validation errors:

```
$response->assertSessionHasNoErrors();
```

assertSessionDoesntHaveErrors

Assert that the session has no validation errors for the given keys:

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null,
$errorBag = 'default');
```

assertSessionMissing

Assert that the session does not contain the given key:

```
$response->assertSessionMissing($key);
```

assertStatus

Assert that the response has a given HTTP status code:

```
$response->assertStatus($code);
```

assertSuccessful

Assert that the response has a successful (>= 200 and < 300) HTTP status code:

```
$response->assertSuccessful();
```

assertUnauthorized

Assert that the response has an unauthorized (401) HTTP status code:

```
$response->assertUnauthorized();
```

assertUnprocessable

Assert that the response has an unprocessable entity (422) HTTP status code:

```
$response->assertUnprocessable();
```

assertValid

Assert that the response has no validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
// Assert that no validation errors are present...
$response->assertValid();

// Assert that the given keys do not have validation errors...
$response->assertValid(['name', 'email']);
```

assertInvalid

Assert that the response has validation errors for the given keys. This method may be used for asserting against responses where the validation errors are returned as a JSON structure or where the validation errors have been flashed to the session:

```
$response->assertInvalid(['name', 'email']);
```

You may also assert that a given key has a particular validation error message. When doing so, you may provide the entire message or only a small portion of the message:

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

assertViewHas

Assert that the response view contains given a piece of data:

```
$response->assertViewHas($key, $value = null);
```

Passing a closure as the second argument to the `assertViewHas` method will allow you to inspect and make assertions against a particular piece of view data:

```
$response->assertViewHas('user', function (User $user) {
    return $user->name === 'Taylor';
});
```

In addition, view data may be accessed as array variables on the response, allowing you to conveniently inspect it:

```
$this->assertEquals('Taylor', $response['name']);
```

assertViewHasAll

Assert that the response view has a given list of data:

```
$response->assertViewHasAll(array $data);
```

This method may be used to assert that the view simply contains data matching the given keys:

```
$response->assertViewHasAll([
    'name',
    'email',
]);
```

Or, you may assert that the view data is present and has specific values:

```
$response->assertViewHasAll([
    'name' => 'Taylor Otwell',
    'email' => 'taylor@example.com,',
]);
```

assertViewIs

Assert that the given view was returned by the route:

```
$response->assertViewIs($value);
```

assertViewMissing

Assert that the given data key was not made available to the view returned in the application's response:

```
$response->assertViewMissing($key);
```

Authentication Assertions

Laravel also provides a variety of authentication related assertions that you may utilize within your application's feature tests. Note that these methods are invoked on the test class itself and not the `Illuminate\Testing\TestResponse` instance returned by methods such as `get` and `post`.

assertAuthenticated

Assert that a user is authenticated:

```
$this->assertAuthenticated($guard = null);
```

assertGuest

Assert that a user is not authenticated:

```
$this->assertGuest($guard = null);
```

assertAuthenticatedAs

Assert that a specific user is authenticated:

```
$this->assertAuthenticatedAs($user, $guard = null);
```


Request Lifecycle

Introduction

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

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

Lifecycle Overview

First Steps

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

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

HTTP / Console Kernels

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

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

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

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

Service Providers

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

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

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

Routing

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

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

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

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

Finishing Up

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

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

Focus On Service Providers

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

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

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

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.

Laravel provides two ways to manage translation strings. First, language strings may be stored in files within the `resources/lang` directory. Within this directory, there may be subdirectories for each language supported by the application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

Or, translation strings may be defined within JSON files that are placed within the `resources/lang` directory. When taking this approach, each language supported by your application would have a corresponding JSON file within this directory. This approach is recommended for application's that have a large number of translatable strings:

```
/resources
    /lang
        en.json
        es.json
```

We'll discuss each approach to managing translation strings within this documentation.

Configuring The Locale

The default language for your application is stored in the `config/app.php` configuration file's `locale` configuration option. You are free to modify this value to suit the needs of your application.

You may modify the default language for a single HTTP request at runtime using the

`setLocale` method provided by the `App` facade:

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

Route::get('/greeting/{locale}', function ($locale) {
    if (! in_array($locale, ['en', 'es', 'fr'])) {
        abort(400);
    }

    App::setLocale($locale);

    //
});
```

You may configure a "fallback language", which will be used when the active language does not contain a given translation string. Like the default language, the fallback language is also configured in the `config/app.php` configuration file:

```
'fallback_locale' => 'en',
```

Determining The Current Locale

You may use the `currentLocale` and `isLocale` methods on the `App` facade to determine the current locale or check if the locale is a given value:

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

$locale = App::currentLocale();

if (App::isLocale('en')) {
    //
}
```

Defining Translation Strings

Using Short Keys

Typically, translation strings are stored in files within the `resources/lang` directory. Within this directory, there should be a subdirectory for each language supported by your application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files return an array of keyed strings. For example:

```
<?php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application!',
];
```

{note} For languages that differ by territory, you should name the language directories according to the ISO 15897. For example, "en_GB" should be used for British English rather than "en-gb".

Using Translation Strings As Keys

For applications with a large number of translatable strings, defining every string with a "short key" can become confusing when referencing the keys in your views and it is

cumbersome to continually invent keys for every translation string supported by your application.

For this reason, Laravel also provides support for defining translation strings using the "default" translation of the string as the key. Translation files that use translation strings as keys are stored as JSON files in the `resources/lang` directory. For example, if your application has a Spanish translation, you should create a `resources/lang/es.json` file:

```
{
    "I love programming.": "Me encanta programar."
}
```

Key / File Conflicts

You should not define translation string keys that conflict with other translation filenames. For example, translating `__('Action')` for the "NL" locale while a `nl/action.php` file exists but a `nl.json` file does not exist will result in the translator returning the contents of `nl/action.php`.

Retrieving Translation Strings

You may retrieve translation strings from your language files using the `__` helper function. If you are using "short keys" to define your translation strings, you should pass the file that contains the key and the key itself to the `__` function using "dot" syntax. For example, let's retrieve the `welcome` translation string from the `resources/lang/en/messages.php` language file:

```
echo __('messages.welcome');
```

If the specified translation string does not exist, the `__` function will return the translation string key. So, using the example above, the `__` function would return `messages.welcome` if the translation string does not exist.

If you are using your default translation strings as your translation keys, you should pass the default translation of your string to the `__` function;

```
echo __('I love programming.');
```

Again, if the translation string does not exist, the `__` function will return the translation string key that it was given.

If you are using the Blade templating engine, you may use the `{{ }}` echo syntax to display the translation string:

```
{{ __('messages.welcome') }}
```

Replacing Parameters In Translation Strings

If you wish, you may define placeholders in your translation strings. All placeholders are prefixed with a `:`. For example, you may define a welcome message with a placeholder name:

```
'welcome' => 'Welcome, :name',
```

To replace the placeholders when retrieving a translation string, you may pass an array of replacements as the second argument to the __ function:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

If your placeholder contains all capital letters, or only has its first letter capitalized, the translated value will be capitalized accordingly:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization; however, Laravel can help you translate strings differently based on pluralization rules that you define. Using a | character, you may distinguish singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

Of course, pluralization is also supported when using translation strings as keys:

```
{
    "There is one apple|There are many apples": "Hay una manzana|Hay muchas manzanas"
}
```

You may even create more complex pluralization rules which specify translation strings for multiple ranges of values:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are
many',
```

After defining a translation string that has pluralization options, you may use the `trans_choice` function to retrieve the line for a given "count". In this example, since the count is greater than one, the plural form of the translation string is returned:

```
echo trans_choice('messages.apples', 10);
```

You may also define placeholder attributes in pluralization strings. These placeholders may be replaced by passing an array as the third argument to the `trans_choice` function:

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',

echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

If you would like to display the integer value that was passed to the `trans_choice` function, you may use the built-in `:count` placeholder:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are
:count',
```

Overriding Package Language Files

Some packages may ship with their own language files. Instead of changing the package's core files to tweak these lines, you may override them by placing files in the `resources/lang/vendor/{package}/{locale}` directory.

So, for example, if you need to override the English translation strings in `messages.php` for a package named `skyrim/hearthfire`, you should place a language file at: `resources/lang/vendor/hearthfire/en/messages.php`. Within this file, you should only define the translation strings you wish to override. Any translation strings you don't override will still be loaded from the package's original language files.

Logging

Introduction

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

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

Under the hood, Laravel utilizes the Monolog library, which provides support for a variety of powerful log handlers. Laravel makes it a cinch to configure these handlers, allowing you to mix and match them to customize your application's log handling.

Configuration

All of the configuration options for your application's logging behavior is housed in the `config/logging.php` configuration file. This file allows you to configure your application's log channels, so be sure to review each of the available channels and their options. We'll review a few common options below.

By default, Laravel will use the `stack` channel when logging messages. The `stack` channel is used to aggregate multiple log channels into a single channel. For more information on building stacks, check out the documentation below.

Configuring The Channel Name

By default, Monolog is instantiated with a "channel name" that matches the current environment, such as `production` or `local`. To change this value, add a `name` option to your channel's configuration:

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

Available Channel Drivers

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

Name	Description
`custom`	A driver that calls a specified factory to create a channel
`daily`	A `RotatingFileHandler` based Monolog driver which rotates daily
`errorlog`	An `ErrorLogHandler` based Monolog driver

Name	Description
monolog	A Monolog factory driver that may use any supported Monolog handler
null	A driver that discards all log messages
papertrail	A SyslogUdpHandler based Monolog driver
single	A single file or path based logger channel (StreamHandler)
slack	A SlackWebhookHandler based Monolog driver
stack	A wrapper to facilitate creating "multi-channel" channels
syslog	A SyslogHandler based Monolog driver

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

Channel Prerequisites

Configuring The Single and Daily Channels

The single and daily channels have three optional configuration options: bubble, permission, and locking.

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

Configuring The Papertrail Channel

The papertrail channel requires the host and port configuration options. You can obtain these values from Papertrail.

Configuring The Slack Channel

The slack channel requires a url configuration option. This URL should match a URL for an

incoming webhook that you have configured for your Slack team.

By default, Slack will only receive logs at the `critical` level and above; however, you can adjust this in your `config/logging.php` configuration file by modifying the `level` configuration option within your Slack log channel's configuration array.

Logging Deprecation Warnings

PHP, Laravel, and other libraries often notify their users that some of their features have been deprecated and will be removed in a future version. If you would like to log these deprecation warnings, you may specify your preferred `deprecations` log channel in your application's `config/logging.php` configuration file:

```
'deprecations' => env('LOG_DEPRECATIONS_CHANNEL', 'null'),
```

Building Log Stacks

As mentioned previously, the `stack` driver allows you to combine multiple channels into a single log channel for convenience. To illustrate how to use log stacks, let's take a look at an example configuration that you might see in a production application:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],

    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],
```

Let's dissect this configuration. First, notice our `stack` channel aggregates two other channels via its `channels` option: `syslog` and `slack`. So, when logging messages, both of these channels will have the opportunity to log the message. However, as we will see below, whether these channels actually log the message may be determined by the message's severity / "level".

Log Levels

Take note of the `level` configuration option present on the `syslog` and `slack` channel configurations in the example above. This option determines the minimum "level" a message must be in order to be logged by the channel. Monolog, which powers Laravel's logging services, offers all of the log levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, and **debug**.

So, imagine we log a message using the `debug` method:

```
Log::debug('An informational message.');
```

Given our configuration, the `syslog` channel will write the message to the system log; however, since the error message is not `critical` or above, it will not be sent to Slack. However, if we log an `emergency` message, it will be sent to both the system log and Slack since the `emergency` level is above our minimum level threshold for both channels:

```
Log::emergency('The system is down!');
```

Writing Log Messages

You may write information to the logs using the `Log` facade. As previously mentioned, the logger provides the eight logging levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info** and **debug**:

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

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

You may call any of these methods to log a message for the corresponding level. By default, the message will be written to the default log channel as configured by your `logging` configuration file:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Log;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        Log::info('Showing the user profile for user: '.$id);

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

Contextual Information

An array of contextual data may be passed to the log methods. This contextual data will be formatted and displayed with the log message:

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

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

Occasionally, you may wish to specify some contextual information that should be included with all subsequent log entries. For example, you may wish to log a request ID that is associated with each incoming request to your application. To accomplish this, you may call the `Log` facade's `withContext` method:

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;

class AssignRequestId
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $requestId = (string) Str::uuid();

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

        return $next($request)->header('Request-Id', $requestId);
    }
}
```

Writing To Specific Channels

Sometimes you may wish to log a message to a channel other than your application's default channel. You may use the `channel` method on the `Log` facade to retrieve and log to any channel defined in your configuration file:

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

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

If you would like to create an on-demand logging stack consisting of multiple channels, you

may use the `stack` method:

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

On-Demand Channels

It is also possible to create an on-demand channel by providing the configuration at runtime without that configuration being present in your application's `logging` configuration file. To accomplish this, you may pass a configuration array to the `Log` facade's `build` method:

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

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

You may also wish to include an on-demand channel in an on-demand logging stack. This can be achieved by including your on-demand channel instance in the array passed to the `stack` method:

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

$channel = Log::build([
  'driver' => 'single',
  'path' => storage_path('logs/custom.log'),
]);

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

Monolog Channel Customization

Customizing Monolog For Channels

Sometimes you may need complete control over how Monolog is configured for an existing channel. For example, you may want to configure a custom Monolog `FormatterInterface` implementation for Laravel's built-in `single` channel.

To get started, define a `tap` array on the channel's configuration. The `tap` array should contain a list of classes that should have an opportunity to customize (or "tap" into) the Monolog instance after it is created. There is no conventional location where these classes should be placed, so you are free to create a directory within your application to contain these classes:

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

Once you have configured the `tap` option on your channel, you're ready to define the class that will customize your Monolog instance. This class only needs a single method: `__invoke`, which receives an `Illuminate\Log\Logger` instance. The `Illuminate\Log\Logger` instance proxies all method calls to the underlying Monolog instance:

```
<?php

namespace App\Logging;

use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    /**
     * Customize the given logger instance.
     *
     * @param  \Illuminate\Log\Logger  $logger
     * @return void
     */
    public function __invoke($logger)
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(new LineFormatter(
                '[%datetime%] %channel%.%level_name%: %message%
%context% %extra%'
            ));
        }
    }
}
```

{tip} All of your "tap" classes are resolved by the service container, so any constructor dependencies they require will automatically be injected.

Creating Monolog Handler Channels

Monolog has a variety of available handlers and Laravel does not include a built-in channel for each one. In some cases, you may wish to create a custom channel that is merely an instance of a specific Monolog handler that does not have a corresponding Laravel log driver. These channels can be easily created using the `monolog` driver.

When using the `monolog` driver, the `handler` configuration option is used to specify which handler will be instantiated. Optionally, any constructor parameters the handler needs may be specified using the `with` configuration option:

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

Monolog Formatters

When using the `monolog` driver, the Monolog `LineFormatter` will be used as the default formatter. However, you may customize the type of formatter passed to the handler using the `formatter` and `formatter_with` configuration options:

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

If you are using a Monolog handler that is capable of providing its own formatter, you may set the value of the `formatter` configuration option to `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

Creating Custom Channels Via Factories

If you would like to define an entirely custom channel in which you have full control over Monolog's instantiation and configuration, you may specify a `custom` driver type in your

`config/logging.php` configuration file. Your configuration should include a `via` option that contains the name of the factory class which will be invoked to create the Monolog instance:

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

Once you have configured the `custom` driver channel, you're ready to define the class that will create your Monolog instance. This class only needs a single `__invoke` method which should return the Monolog logger instance. The method will receive the channels configuration array as its only argument:

```
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Create a custom Monolog instance.
     *
     * @param  array  $config
     * @return \Monolog\Logger
     */
    public function __invoke(array $config)
    {
        return new Logger(...);
    }
}
```

Middleware

Introduction

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

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

Defining Middleware

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

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

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

"pass"), you should call the `$next` callback with the `$request`.

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

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

Middleware & Responses

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

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

```php
<?php

namespace App\Http\Middleware;

use Closure;

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

        // Perform action

        return $response;
    }
}
```

Registering Middleware

Global Middleware

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

Assigning Middleware To Routes

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

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

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

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

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

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

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

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

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

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

Excluding Middleware

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

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

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

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

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

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

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

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

Middleware Groups

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

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

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

    'api' => [
        'throttle:api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

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

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

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

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

Sorting Middleware

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

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

Middleware Parameters

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        return $next($request);
    }

}
```

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

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

Terminable Middleware

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

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

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

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

The `terminate` method should receive both the request and the response. Once you have

defined a terminable middleware, you should add it to the list of routes or global middleware in the `app/Http/Kernel.php` file.

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

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

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

Database: Migrations

Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

Generating Migrations

You may use the `make:migration` Artisan command to generate a database migration. The new migration will be placed in your `database/migrations` directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

Laravel will use the name of the migration to attempt to guess the name of the table and whether or not the migration will be creating a new table. If Laravel is able to determine the table name from the migration name, Laravel will pre-fill the generated migration file with the specified table. Otherwise, you may simply specify the table in the migration file manually.

If you would like to specify a custom path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The given path should be relative to your application's base path.

{tip} Migration stubs may be customized using stub publishing.

Squashing Migrations

As you build your application, you may accumulate more and more migrations over time. This can lead to your `database/migrations` directory becoming bloated with potentially hundreds of migrations. If you would like, you may "squash" your migrations into a single SQL file. To get started, execute the `schema:dump` command:

```
php artisan schema:dump

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

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

You should commit your database schema file to source control so that other new developers on your team may quickly create your application's initial database structure.

{note} Migration squashing is only available for the MySQL, PostgreSQL, and SQLite databases and utilizes the database's command-line client. Schema dumps may not be restored to in-memory SQLite databases.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should reverse the operations performed by the `up` method.

Within both of these methods, you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, check out its documentation. For example, the following migration creates a `flights` table:

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Anonymous Migrations

As you may have noticed in the example above, Laravel will automatically assign a class name to all of the migrations that you generate using the `make:migration` command. However, if you wish, you may return an anonymous class from your migration file. This is primarily useful if your application accumulates many migrations and two of them have a class name collision:

```php
<?php

use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    //
};
```

Setting The Migration Connection

If your migration will be interacting with a database connection other than your application's default database connection, you should set the `$connection` property of your migration:

```php
/**
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    //
}
```

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

If you would like to see which migrations have run thus far, you may use the `migrate:status` Artisan command:

```
php artisan migrate:status
```

Forcing Migrations To Run In Production

Some migration operations are destructive, which means they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before the commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To roll back the latest migration operation, you may use the `rollback` Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may roll back a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will roll back the last five migrations:

```
php artisan migrate:rollback --step=5
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Roll Back & Migrate Using A Single Command

The `migrate:refresh` command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

You may roll back and re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will roll back and re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Tables & Migrate

The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

{note} The `migrate:fresh` command will drop all database tables regardless of their prefix. This command should be used with caution when developing on a database that is shared with other applications.

Tables

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments: the first is the name of the table, while the second is a closure which receives a `Blueprint` object that may be used to define the new table:

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

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

When creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

Database Connection & Table Options

If you want to perform a schema operation on a database connection that is not your application's default connection, use the `connection` method:

```
Schema::connection('sqlite')->create('users', function (Blueprint
$table) {
    $table->id();
});
```

In addition, a few other properties and methods may be used to define other aspects of the table's creation. The `engine` property may be used to specify the table's storage engine when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    // ...
});
```

The `charset` and `collation` properties may be used to specify the character set and collation for the created table when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

The `temporary` method may be used to indicate that the table should be "temporary". Temporary tables are only visible to the current connection's database session and are dropped automatically when the connection is closed:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

Updating Tables

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives a `Blueprint` instance you may use to add columns or indexes to the table:

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

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

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

Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Renaming Tables With Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a closure that receives an `Illuminate\Database\Schema\Blueprint` instance you may use to add columns to the table:

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

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Available Column Types

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

[bigIncrements](#column-method-bigIncrements) [bigInteger](#column-method-bigInteger) [binary](#column-method-binary) [boolean](#column-method-boolean) [char](#column-method-char) [dateTimeTz](#column-method-dateTimeTz) [dateTime](#column-method-dateTime) [date](#column-method-date) [decimal](#column-method-decimal) [double](#column-method-double) [enum](#column-method-enum) [float](#column-method-float) [foreignId](#column-method-foreignId) [foreignIdFor](#column-method-foreignIdFor) [foreignUuid](#column-method-foreignUuid) [geometryCollection](#column-method-geometryCollection) [geometry](#column-method-geometry) [id](#column-method-id) [increments](#column-method-increments) [integer](#column-method-integer) [ipAddress](#column-method-ipAddress) [json](#column-method-json) [jsonb](#column-method-jsonb) [lineString](#column-method-lineString) [longText](#column-method-longText) [macAddress](#column-method-macAddress) [mediumIncrements](#column-method-mediumIncrements) [mediumInteger](#column-method-mediumInteger) [mediumText](#column-method-mediumText) [morphs](#column-method-morphs) [multiLineString](#column-method-multiLineString) [multiPoint](#column-method-

multiPoint) [multiPolygon](#column-method-multiPolygon) [nullableMorphs](#column-method-nullableMorphs) [nullableTimestamps](#column-method-nullableTimestamps) [nullableUuidMorphs](#column-method-nullableUuidMorphs) [point](#column-method-point) [polygon](#column-method-polygon) [rememberToken](#column-method-rememberToken) [set](#column-method-set) [smallIncrements](#column-method-smallIncrements) [smallInteger](#column-method-smallInteger) [softDeletesTz](#column-method-softDeletesTz) [softDeletes](#column-method-softDeletes) [string](#column-method-string) [text](#column-method-text) [timeTz](#column-method-timeTz) [time](#column-method-time) [timestampTz](#column-method-timestampTz) [timestamp](#column-method-timestamp) [timestampsTz](#column-method-timestampsTz) [timestamps](#column-method-timestamps) [tinyIncrements](#column-method-tinyIncrements) [tinyInteger](#column-method-tinyInteger) [tinyText](#column-method-tinyText) [unsignedBigInteger](#column-method-unsignedBigInteger) [unsignedDecimal](#column-method-unsignedDecimal) [unsignedInteger](#column-method-unsignedInteger) [unsignedMediumInteger](#column-method-unsignedMediumInteger) [unsignedSmallInteger](#column-method-unsignedSmallInteger) [unsignedTinyInteger](#column-method-unsignedTinyInteger) [uuidMorphs](#column-method-uuidMorphs) [uuid](#column-method-uuid) [year](#column-method-year)

`bigIncrements()` {.collection-method .first-collection-method}

The `bigIncrements` method creates an auto-incrementing `UNSIGNED BIGINT` (primary key) equivalent column:

```
$table->bigIncrements('id');
```

`bigInteger()` {.collection-method}

The `bigInteger` method creates a `BIGINT` equivalent column:

```
$table->bigInteger('votes');
```

`binary()` {.collection-method}

The `binary` method creates a `BLOB` equivalent column:

```
$table->binary('photo');
```

`boolean()` {.collection-method}

The `boolean` method creates a `BOOLEAN` equivalent column:

```
$table->boolean('confirmed');
```

`char()` {.collection-method}

The `char` method creates a `CHAR` equivalent column with of a given length:

```
$table->char('name', 100);
```

`dateTimeTz()` {.collection-method}

The `dateTimeTz` method creates a `DATETIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->dateTimeTz('created_at', $precision = 0);
```

`dateTime()` {.collection-method}

The `dateTime` method creates a `DATETIME` equivalent column with an optional precision (total digits):

```
$table->dateTime('created_at', $precision = 0);
```

`date()` {.collection-method}

The `date` method creates a `DATE` equivalent column:

```
$table->date('created_at');
```

`decimal()` {.collection-method}

The `decimal` method creates a `DECIMAL` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

`double()` {.collection-method}

The `double` method creates a `DOUBLE` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->double('amount', 8, 2);
```

`enum()` {.collection-method}

The `enum` method creates a `ENUM` equivalent column with the given valid values:

```
$table->enum('difficulty', ['easy', 'hard']);
```

`float()` {.collection-method}

The `float` method creates a `FLOAT` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->float('amount', 8, 2);
```

`foreignId()` {.collection-method}

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->foreignId('user_id');
```

`foreignIdFor()` {.collection-method}

The `foreignIdFor` method adds a `{column}_id UNSIGNED BIGINT` equivalent column for a given model class:

```
$table->foreignIdFor(User::class);
```

`foreignUuid()` {.collection-method}

The `foreignUuid` method creates a `UUID` equivalent column:

```
$table->foreignUuid('user_id');
```

`geometryCollection()` {.collection-method}

The `geometryCollection` method creates a `GEOMETRYCOLLECTION` equivalent column:

```
$table->geometryCollection('positions');
```

`geometry()` {.collection-method}

The `geometry` method creates a `GEOMETRY` equivalent column:

```
$table->geometry('positions');
```

`id()` {.collection-method}

The `id` method is an alias of the `bigIncrements` method. By default, the method will create an `id` column; however, you may pass a column name if you would like to assign a different name to the column:

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

`increments()` {.collection-method}

The `increments` method creates an auto-incrementing `UNSIGNED INTEGER` equivalent column as a primary key:

```
$table->increments('id');
```

`integer()` {.collection-method}

The `integer` method creates an `INTEGER` equivalent column:

```
$table->integer('votes');
```

`ipAddress()` {.collection-method}

The `ipAddress` method creates a `VARCHAR` equivalent column:

```
$table->ipAddress('visitor');
```

`json()` {.collection-method}

The `json` method creates a `JSON` equivalent column:

```
$table->json('options');
```

`jsonb()` {.collection-method}

The `jsonb` method creates a `JSONB` equivalent column:

```
$table->jsonb('options');
```

`lineString()` {.collection-method}

The `lineString` method creates a `LINESTRING` equivalent column:

```
$table->lineString('positions');
```

`longText()` {.collection-method}

The `longText` method creates a `LONGTEXT` equivalent column:

```
$table->longText('description');
```

`macAddress()` {.collection-method}

The `macAddress` method creates a column that is intended to hold a MAC address. Some database systems, such as PostgreSQL, have a dedicated column type for this type of data. Other database systems will use a string equivalent column:

```
$table->macAddress('device');
```

`mediumIncrements()` {.collection-method}

The `mediumIncrements` method creates an auto-incrementing `UNSIGNED MEDIUMINT` equivalent column as a primary key:

```
$table->mediumIncrements('id');
```

`mediumInteger()` {.collection-method}

The `mediumInteger` method creates a `MEDIUMINT` equivalent column:

```
$table->mediumInteger('votes');
```

`mediumText()` {.collection-method}

The `mediumText` method creates a `MEDIUMTEXT` equivalent column:

```
$table->mediumText('description');
```

`morphs()` {.collection-method}

The `morphs` method is a convenience method that adds a `{column}_id UNSIGNED BIGINT` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->morphs('taggable');
```

`multiLineString()` {.collection-method}

The `multiLineString` method creates a `MULTILINESTRING` equivalent column:

```
$table->multiLineString('positions');
```

`multiPoint()` {.collection-method}

The `multiPoint` method creates a `MULTIPOINT` equivalent column:

```
$table->multiPoint('positions');
```

`multiPolygon()` {.collection-method}

The `multiPolygon` method creates a `MULTIPOLYGON` equivalent column:

```
$table->multiPolygon('positions');
```

`nullableTimestamps()` {.collection-method}

The `nullableTimestamps` method is an alias of the timestamps method:

```
$table->nullableTimestamps(0);
```

`nullableMorphs()` {.collection-method}

The method is similar to the morphs method; however, the columns that are created will be "nullable":

```
$table->nullableMorphs('taggable');
```

`nullableUuidMorphs()` {.collection-method}

The method is similar to the uuidMorphs method; however, the columns that are created will be "nullable":

```
$table->nullableUuidMorphs('taggable');
```

`point()` {.collection-method}

The `point` method creates a `POINT` equivalent column:

```
$table->point('position');
```

`polygon()` {.collection-method}

The `polygon` method creates a `POLYGON` equivalent column:

```
$table->polygon('position');
```

`rememberToken()` {.collection-method}

The `rememberToken` method creates a nullable, `VARCHAR(100)` equivalent column that is intended to store the current "remember me" authentication token:

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

`set()` {.collection-method}

The `set` method creates a `SET` equivalent column with the given list of valid values:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

`smallIncrements()` {.collection-method}

The `smallIncrements` method creates an auto-incrementing `UNSIGNED SMALLINT` equivalent column as a primary key:

```
$table->smallIncrements('id');
```

`smallInteger()` {.collection-method}

The `smallInteger` method creates a `SMALLINT` equivalent column:

```
$table->smallInteger('votes');
```

`softDeletesTz()` {.collection-method}

The `softDeletesTz` method adds a nullable `deleted_at TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits). This column is intended to store

the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

`softDeletes()` {.collection-method}

The `softDeletes` method adds a nullable `deleted_at` `TIMESTAMP` equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

`string()` {.collection-method}

The `string` method creates a `VARCHAR` equivalent column of the given length:

```
$table->string('name', 100);
```

`text()` {.collection-method}

The `text` method creates a `TEXT` equivalent column:

```
$table->text('description');
```

`timeTz()` {.collection-method}

The `timeTz` method creates a `TIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timeTz('sunrise', $precision = 0);
```

`time()` {.collection-method}

The `time` method creates a `TIME` equivalent column with an optional precision (total digits):

```
$table->time('sunrise', $precision = 0);
```

`timestampTz()` {.collection-method}

The `timestampTz` method creates a `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timestampTz('added_at', $precision = 0);
```

`timestamp()` {.collection-method}

The `timestamp` method creates a `TIMESTAMP` equivalent column with an optional precision (total digits):

```
$table->timestamp('added_at', $precision = 0);
```

`timestampsTz()` {.collection-method}

The `timestampsTz` method creates `created_at` and `updated_at` `TIMESTAMP` (with timezone) equivalent columns with an optional precision (total digits):

```
$table->timestampsTz($precision = 0);
```

`timestamps()` {.collection-method}

The `timestamps` method creates `created_at` and `updated_at` `TIMESTAMP` equivalent columns with an optional precision (total digits):

```
$table->timestamps($precision = 0);
```

`tinyIncrements()` {.collection-method}

The `tinyIncrements` method creates an auto-incrementing `UNSIGNED TINYINT` equivalent column as a primary key:

```
$table->tinyIncrements('id');
```

`tinyInteger()` {.collection-method}

The `tinyInteger` method creates a `TINYINT` equivalent column:

```
$table->tinyInteger('votes');
```

`tinyText()` {.collection-method}

The `tinyText` method creates a `TINYTEXT` equivalent column:

```
$table->tinyText('notes');
```

`unsignedBigInteger()` {.collection-method}

The `unsignedBigInteger` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->unsignedBigInteger('votes');
```

`unsignedDecimal()` {.collection-method}

The `unsignedDecimal` method creates an `UNSIGNED DECIMAL` equivalent column with an optional precision (total digits) and scale (decimal digits):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

`unsignedInteger()` {.collection-method}

The `unsignedInteger` method creates an `UNSIGNED INTEGER` equivalent column:

```
$table->unsignedInteger('votes');
```

`unsignedMediumInteger()` {.collection-method}

The `unsignedMediumInteger` method creates an `UNSIGNED MEDIUMINT` equivalent column:

```
$table->unsignedMediumInteger('votes');
```

`unsignedSmallInteger()` {.collection-method}

The `unsignedSmallInteger` method creates an `UNSIGNED SMALLINT` equivalent column:

```
$table->unsignedSmallInteger('votes');
```

`unsignedTinyInteger()` {.collection-method}

The `unsignedTinyInteger` method creates an `UNSIGNED TINYINT` equivalent column:

```
$table->unsignedTinyInteger('votes');
```

`uuidMorphs()` {.collection-method}

The `uuidMorphs` method is a convenience method that adds a `{column}_id CHAR(36)` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship that use UUID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->uuidMorphs('taggable');
```

`uuid()` {.collection-method}

The `uuid` method creates a `UUID` equivalent column:

```
$table->uuid('id');
```

`year()` {.collection-method}

The `year` method creates a `YEAR` equivalent column:

```
$table->year('birth_year');
```

Column Modifiers

In addition to the column types listed above, there are several column "modifiers" you may use when adding a column to a database table. For example, to make the column "nullable", you may use the `nullable` method:

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

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

The following table contains all of the available column modifiers. This list does not include index modifiers:

Modifier	Description
`->after('column')`	Place the column "after" another column (MySQL).
`->autoIncrement()`	Set INTEGER columns as auto-incrementing (primary key).
`->charset('utf8mb4')`	Specify a character set for the column (MySQL).
`->collation('utf8mb4_unicode_ci')`	Specify a collation for the column (MySQL/PostgreSQL/SQL Server).
`->comment('my comment')`	Add a comment to a column (MySQL/PostgreSQL).
`->default($value)`	Specify a "default" value for the column.
`->first()`	Place the column "first" in the table (MySQL).
`->from($integer)`	Set the starting value of an auto-incrementing field (MySQL / PostgreSQL).
`->invisible()`	Make the column "invisible" to `SELECT *` queries (MySQL).
`->nullable($value = true)`	Allow NULL values to be inserted into the column.
`->storedAs($expression)`	Create a stored generated column (MySQL / PostgreSQL).
`->unsigned()`	Set INTEGER columns as UNSIGNED (MySQL).
`->useCurrent()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value.
`->useCurrentOnUpdate()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP when a record is updated.
`->virtualAs($expression)`	Create a virtual generated column (MySQL).
`->generatedAs($expression)`	Create an identity column with specified sequence options (PostgreSQL).
`->always()`	Defines the precedence of sequence values over input for an identity column (PostgreSQL).

Modifier	Description
`->isGeometry()`	Set spatial column type to `geometry` - the default type is `geography` (PostgreSQL).

Default Expressions

The `default` modifier accepts a value or an `Illuminate\Database\Query\Expression` instance. Using an `Expression` instance will prevent Laravel from wrapping the value in quotes and allow you to use database specific functions. One situation where this is particularly useful is when you need to assign default values to JSON columns:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new
Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
}
```

{note} Support for default expressions depends on your database driver, database version, and the field type. Please refer to your database's documentation.

Column Order

When using the MySQL database, the `after` method may be used to add columns after an existing column in the schema:

```
$table->after('password', function ($table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Modifying Columns

Prerequisites

Before modifying a column, you must install the `doctrine/dbal` package using the Composer package manager. The Doctrine DBAL library is used to determine the current state of the column and to create the SQL queries needed to make the requested changes to your column:

```
composer require doctrine/dbal
```

If you plan to modify columns created using the `timestamp` method, you must also add the following configuration to your application's `config/database.php` configuration file:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

{note} If your application is using Microsoft SQL Server, please ensure that you install `doctrine/dbal:^3.0`.

Updating Column Attributes

The `change` method allows you to modify the type and attributes of existing columns. For example, you may wish to increase the size of a `string` column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50. To accomplish this, we simply define the new state of the column and then call the `change` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

We could also modify a column to be nullable:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

{note} The following column types can be modified: `bigInteger`, `binary`, `boolean`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger`, and `uuid`. To modify a `timestamp` column type a Doctrine type must be registered.

Renaming Columns

To rename a column, you may use the `renameColumn` method provided by the schema builder blueprint. Before renaming a column, ensure that you have installed the `doctrine/dbal` library via the Composer package manager:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

{note} Renaming an `enum` column is not currently supported.

Dropping Columns

To drop a column, you may use the `dropColumn` method on the schema builder blueprint. If your application is utilizing an SQLite database, you must install the `doctrine/dbal` package via the Composer package manager before the `dropColumn` method may be used:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

You may drop multiple columns from a table by passing an array of column names to the `dropColumn` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

{note} Dropping or modifying multiple columns within a single migration while using an SQLite database is not supported.

Available Command Aliases

Laravel provides several convenient methods related to dropping common types of columns. Each of these methods is described in the table below:

Command	Description
`$table->dropMorphs('morphable');`	Drop the `morphable_id` and `morphable_type` columns.
`$table->dropRememberToken();`	Drop the `remember_token` column.
`$table->dropSoftDeletes();`	Drop the `deleted_at` column.
`$table->dropSoftDeletesTz();`	Alias of `dropSoftDeletes()` method.
`$table->dropTimestamps();`	Drop the `created_at` and `updated_at` columns.
`$table->dropTimestampsTz();`	Alias of `dropTimestamps()` method.

Indexes

Creating Indexes

The Laravel schema builder supports several types of indexes. The following example creates a new `email` column and specifies that its values should be unique. To create the index, we can chain the `unique` method onto the column definition:

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

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
});
```

Alternatively, you may create the index after defining the column. To do so, you should call the `unique` method on the schema builder blueprint. This method accepts the name of the column that should receive a unique index:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound (or composite) index:

```
$table->index(['account_id', 'created_at']);
```

When creating an index, Laravel will automatically generate an index name based on the table, column names, and the index type, but you may pass a second argument to the method to specify the index name yourself:

```
$table->unique('email', 'unique_email');
```

Available Index Types

Laravel's schema builder blueprint class provides methods for creating each type of index supported by Laravel. Each index method accepts an optional second argument to specify the name of the index. If omitted, the name will be derived from the names of the table and column(s) used for the index, as well as the index type. Each of the available index methods is described in the table below:

Command	Description
`$table->primary('id');`	Adds a primary key.
`$table->primary(['id', 'parent_id']);`	Adds composite keys.
`$table->unique('email');`	Adds a unique index.
`$table->index('state');`	Adds an index.
`$table->fulltext('body');`	Adds a fulltext index (MySQL/PostgreSQL).
`$table->fulltext('body')->language('english');`	Adds a fulltext index of the specified language (PostgreSQL).
`$table->spatialIndex('location');`	Adds a spatial index (except SQLite).

Index Lengths & MySQL / MariaDB

By default, Laravel uses the `utf8mb4` character set. If you are running a version of MySQL older than the 5.7.7 release or MariaDB older than the 10.2.2 release, you may need to manually configure the default string length generated by migrations in order for MySQL to create indexes for them. You may configure the default string length by calling the `Schema::defaultStringLength` method within the `boot` method of your `App\Providers\AppServiceProvider` class:

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

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

Alternatively, you may enable the `innodb_large_prefix` option for your database. Refer to your database's documentation for instructions on how to properly enable this option.

Renaming Indexes

To rename an index, you may use the `renameIndex` method provided by the schema builder blueprint. This method accepts the current index name as its first argument and the desired name as its second argument:

```
$table->renameIndex('from', 'to')
```

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns an index name based on the table name, the name of the indexed column, and the index type. Here are some examples:

Command	Description
`$table->dropPrimary('users_id_primary');`	Drop a primary key from the "users" table.
`$table->dropUnique('users_email_unique');`	Drop a unique index from the "users" table.

Command	Description
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.
`$table->dropSpatialIndex('geo_location_spatialindex');`	Drop a spatial index from the "geo" table (except SQLite).

If you pass an array of columns into a method that drops indexes, the conventional index name will be generated based on the table name, columns, and index type:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

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

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

Since this syntax is rather verbose, Laravel provides additional, terser methods that use conventions to provide a better developer experience. When using the `foreignId` method to create your column, the example above can be rewritten like so:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column, while the `constrained` method will use conventions to determine the table and column name being referenced. If your table name does not match Laravel's conventions, you may specify the table name by passing it as an argument to the `constrained` method:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

An alternative, expressive syntax is also provided for these actions:

Method	Description
`$table->cascadeOnUpdate();`	Updates should cascade.
`$table->restrictOnUpdate();`	Updates should be restricted.
`$table->cascadeOnDelete();`	Deletes should cascade.
`$table->restrictOnDelete();`	Deletes should be restricted.
`$table->nullOnDelete();`	Deletes should set the foreign key value to null.

Any additional column modifiers must be called before the `constrained` method:

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

Dropping Foreign Keys

To drop a foreign key, you may use the `dropForeign` method, passing the name of the foreign key constraint to be deleted as an argument. Foreign key constraints use the same naming convention as indexes. In other words, the foreign key constraint name is based on the name of the table and the columns in the constraint, followed by a "_foreign" suffix:

```
$table->dropForeign('posts_user_id_foreign');
```

Alternatively, you may pass an array containing the column name that holds the foreign key to the `dropForeign` method. The array will be converted to a foreign key constraint name using Laravel's constraint naming conventions:

```
$table->dropForeign(['user_id']);
```

Toggling Foreign Key Constraints

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

{note} SQLite disables foreign key constraints by default. When using SQLite, make sure to enable foreign key support in your database configuration before attempting to create them in your migrations. In addition, SQLite only supports foreign keys upon creation of the table and not when tables are altered.

Events

For convenience, each migration operation will dispatch an event. All of the following events extend the base `Illuminate\Database\Events\MigrationEvent` class:

Class	Description
`Illuminate\Database\Events\MigrationsStarted`	A batch of migrations is about to be executed.
`Illuminate\Database\Events\MigrationsEnded`	A batch of migrations has finished executing.
`Illuminate\Database\Events\MigrationStarted`	A single migration is about to be executed.
`Illuminate\Database\Events\MigrationEnded`	A single migration has finished executing.

Compiling Assets (Mix)

Introduction

Laravel Mix, a package developed by Laracasts creator Jeffrey Way, provides a fluent API for defining webpack build steps for your Laravel application using several common CSS and JavaScript pre-processors.

In other words, Mix makes it a cinch to compile and minify your application's CSS and JavaScript files. Through simple method chaining, you can fluently define your asset pipeline. For example:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css');
```

If you've ever been confused and overwhelmed about getting started with webpack and asset compilation, you will love Laravel Mix. However, you are not required to use it while developing your application; you are free to use any asset pipeline tool you wish, or even none at all.

{tip} If you need a head start building your application with Laravel and Tailwind CSS, check out one of our application starter kits.

Installation & Setup

Installing Node

Before running Mix, you must first ensure that Node.js and NPM are installed on your machine:

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

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

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

Installing Laravel Mix

The only remaining step is to install Laravel Mix. Within a fresh installation of Laravel, you'll find a `package.json` file in the root of your directory structure. The default `package.json` file already includes everything you need to get started using Laravel Mix. Think of this file like your `composer.json` file, except it defines Node dependencies instead of PHP dependencies. You may install the dependencies it references by running:

```
npm install
```

Running Mix

Mix is a configuration layer on top of webpack, so to run your Mix tasks you only need to execute one of the NPM scripts that are included in the default Laravel `package.json` file. When you run the `dev` or `production` scripts, all of your application's CSS and JavaScript assets will be compiled and placed in your application's `public` directory:

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run prod
```

Watching Assets For Changes

The `npm run watch` command will continue running in your terminal and watch all relevant CSS and JavaScript files for changes. Webpack will automatically recompile your assets when it detects a change to one of these files:

```
npm run watch
```

Webpack may not be able to detect your file changes in certain local development environments. If this is the case on your system, consider using the `watch-poll` command:

```
npm run watch-poll
```

Working With Stylesheets

Your application's `webpack.mix.js` file is your entry point for all asset compilation. Think of it as a light configuration wrapper around webpack. Mix tasks can be chained together to define exactly how your assets should be compiled.

Tailwind CSS

Tailwind CSS is a modern, utility-first framework for building amazing sites without ever leaving your HTML. Let's dig into how to start using it in a Laravel project with Laravel Mix. First, we should install Tailwind using NPM and generate our Tailwind configuration file:

```
npm install

npm install -D tailwindcss

npx tailwindcss init
```

The `init` command will generate a `tailwind.config.js` file. Within this file, you may configure the paths to all of your application's templates and JavaScript so that Tailwind can tree-shake unused styles when optimizing your CSS for production:

```
purge: [
    './storage/framework/views/*.php',
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
],
```

Next, you should add each of Tailwind's "layers" to your application's `resources/css/app.css` file:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Once you have configured Tailwind's layers, you are ready to update your application's `webpack.mix.js` file to compile your Tailwind powered CSS:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css', [
        require('tailwindcss'),
    ]);
```

Finally, you should reference your stylesheet in your application's primary layout template. Many applications choose to store this template at `resources/views/layouts/app.blade.php`. In addition, ensure you add the responsive viewport `meta` tag if it's not already present:

```
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <link href="/css/app.css" rel="stylesheet">
</head>
```

PostCSS

PostCSS, a powerful tool for transforming your CSS, is included with Laravel Mix out of the box. By default, Mix leverages the popular Autoprefixer plugin to automatically apply all necessary CSS3 vendor prefixes. However, you're free to add any additional plugins that are appropriate for your application.

First, install the desired plugin through NPM and include it in your array of plugins when calling Mix's `postCss` method. The `postCss` method accepts the path to your CSS file as its first argument and the directory where the compiled file should be placed as its second argument:

```
mix.postCss('resources/css/app.css', 'public/css', [
    require('postcss-custom-properties')
]);
```

Or, you may execute `postCss` with no additional plugins in order to achieve simple CSS compilation and minification:

```
mix.postCss('resources/css/app.css', 'public/css');
```

Sass

The `sass` method allows you to compile Sass into CSS that can be understood by web browsers. The `sass` method accepts the path to your Sass file as its first argument and the directory where the compiled file should be placed as its second argument:

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

You may compile multiple Sass files into their own respective CSS files and even customize the output directory of the resulting CSS by calling the `sass` method multiple times:

```
mix.sass('resources/sass/app.sass', 'public/css')
    .sass('resources/sass/admin.sass', 'public/css/admin');
```

URL Processing

Because Laravel Mix is built on top of webpack, it's important to understand a few webpack concepts. For CSS compilation, webpack will rewrite and optimize any `url()` calls within your stylesheets. While this might initially sound strange, it's an incredibly powerful piece of functionality. Imagine that we want to compile Sass that includes a relative URL to an image:

```
.example {
    background: url('../images/example.png');
}
```

{note} Absolute paths for any given `url()` will be excluded from URL-rewriting. For example, `url('/images/thing.png')` or `url('http://example.com/images/thing.png')` won't be modified.

By default, Laravel Mix and webpack will find `example.png`, copy it to your `public/images` folder, and then rewrite the `url()` within your generated stylesheet. As such, your compiled CSS will be:

```
.example {
    background:
url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);
}
```

As useful as this feature may be, your existing folder structure may already be configured in a way you like. If this is the case, you may disable `url()` rewriting like so:

```
mix.sass('resources/sass/app.scss', 'public/css').options({
    processCssUrls: false
});
```

With this addition to your `webpack.mix.js` file, Mix will no longer match any `url()` or copy assets to your public directory. In other words, the compiled CSS will look just like how you originally typed it:

```
.example {
    background: url("../images/thing.png");
}
```

Source Maps

Though disabled by default, source maps may be activated by calling the

`mix.sourceMaps()` method in your `webpack.mix.js` file. Though it comes with a compile/performance cost, this will provide extra debugging information to your browser's developer tools when using compiled assets:

```
mix.js('resources/js/app.js', 'public/js')
    .sourceMaps();
```

Style Of Source Mapping

Webpack offers a variety of source mapping styles. By default, Mix's source mapping style is set to `eval-source-map`, which provides a fast rebuild time. If you want to change the mapping style, you may do so using the `sourceMaps` method:

```
let productionSourceMaps = false;

mix.js('resources/js/app.js', 'public/js')
    .sourceMaps(productionSourceMaps, 'source-map');
```

Working With JavaScript

Mix provides several features to help you work with your JavaScript files, such as compiling modern ECMAScript, module bundling, minification, and concatenating plain JavaScript files. Even better, this all works seamlessly, without requiring an ounce of custom configuration:

```
mix.js('resources/js/app.js', 'public/js');
```

With this single line of code, you may now take advantage of:

- The latest EcmaScript syntax. - Modules - Minification for production environments.

Vue

Mix will automatically install the Babel plugins necessary for Vue single-file component compilation support when using the `vue` method. No further configuration is required:

```
mix.js('resources/js/app.js', 'public/js')
   .vue();
```

Once your JavaScript has been compiled, you can reference it in your application:

```
<head>
    <!-- ... -->

    <script src="/js/app.js"></script>
</head>
```

React

Mix can automatically install the Babel plugins necessary for React support. To get started, add a call to the `react` method:

```
mix.js('resources/js/app.jsx', 'public/js')
   .react();
```

Behind the scenes, Mix will download and include the appropriate `babel-preset-react` Babel plugin. Once your JavaScript has been compiled, you can reference it in your application:

```
<head>
    <!-- ... -->

    <script src="/js/app.js"></script>
</head>
```

Vendor Extraction

One potential downside to bundling all of your application-specific JavaScript with your vendor libraries such as React and Vue is that it makes long-term caching more difficult. For example, a single update to your application code will force the browser to re-download all of your vendor libraries even if they haven't changed.

If you intend to make frequent updates to your application's JavaScript, you should consider extracting all of your vendor libraries into their own file. This way, a change to your application code will not affect the caching of your large `vendor.js` file. Mix's `extract` method makes this a breeze:

```
mix.js('resources/js/app.js', 'public/js')
    .extract(['vue'])
```

The `extract` method accepts an array of all libraries or modules that you wish to extract into a `vendor.js` file. Using the snippet above as an example, Mix will generate the following files:

- `public/js/manifest.js`: *The Webpack manifest runtime* - `public/js/vendor.js`: *Your vendor libraries* - `public/js/app.js`: *Your application code*

To avoid JavaScript errors, be sure to load these files in the proper order:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

Custom Webpack Configuration

Occasionally, you may need to manually modify the underlying Webpack configuration. For example, you might have a special loader or plugin that needs to be referenced.

Mix provides a useful `webpackConfig` method that allows you to merge any short Webpack configuration overrides. This is particularly appealing, 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')
        ]
    }
});
```

Versioning / Cache Busting

Many developers suffix their compiled assets with a timestamp or unique token to force browsers to load the fresh assets instead of serving stale copies of the code. Mix can automatically handle this for you using the `version` method.

The `version` method will append a unique hash to the filenames of all compiled files, allowing for more convenient cache busting:

```
mix.js('resources/js/app.js', 'public/js')
    .version();
```

After generating the versioned file, you won't know the exact filename. So, you should use Laravel's global `mix` function within your views to load the appropriately hashed asset. The `mix` function will automatically determine the current name of the hashed file:

```
<script src="{{ mix('/js/app.js') }}"></script>
```

Because versioned files are usually unnecessary in development, you may instruct the versioning process to only run during `npm run prod`:

```
mix.js('resources/js/app.js', 'public/js');

if (mix.inProduction()) {
    mix.version();
}
```

Custom Mix Base URLs

If your Mix compiled assets are deployed to a CDN separate from your application, you will need to change the base URL generated by the `mix` function. You may do so by adding a `mix_url` configuration option to your application's `config/app.php` configuration file:

```
'mix_url' => env('MIX_ASSET_URL', null)
```

After configuring the Mix URL, The `mix` function will prefix the configured URL when generating URLs to assets:

```
https://cdn.example.com/js/app.js?id=1964becbdd96414518cd
```

Browsersync Reloading

BrowserSync can automatically monitor your files for changes, and inject your changes into the browser without requiring a manual refresh. You may enable support for this by calling the `mix.browserSync()` method:

```
mix.browserSync('laravel.test');
```

BrowserSync options may be specified by passing a JavaScript object to the `browserSync` method:

```
mix.browserSync({
    proxy: 'laravel.test'
});
```

Next, start webpack's development server using the `npm run watch` command. Now, when you modify a script or PHP file you can watch as the browser instantly refreshes the page to reflect your changes.

Environment Variables

You may inject environment variables into your `webpack.mix.js` script by prefixing one of the environment variables in your `.env` file with `MIX_`:

```
MIX_SENTRY_DSN_PUBLIC=http://example.com
```

After the variable has been defined in your `.env` file, you may access it via the `process.env` object. However, you will need to restart the task if the environment variable's value changes while the task is running:

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

Notifications

When available, Mix will automatically display OS notifications when compiling, giving you instant feedback as to whether the compilation was successful or not. However, there may be instances when you would prefer to disable these notifications. One such example might be triggering Mix on your production server. Notifications may be deactivated using the `disableNotifications` method:

```
mix.disableNotifications();
```

Mocking

Introduction

When testing Laravel applications, you may wish to "mock" certain aspects of your application so they are not actually executed during a given test. For example, when testing a controller that dispatches an event, you may wish to mock the event listeners so they are not actually executed during the test. This allows you to only test the controller's HTTP response without worrying about the execution of the event listeners since the event listeners can be tested in their own test case.

Laravel provides helpful methods for mocking events, jobs, and other facades out of the box. These helpers primarily provide a convenience layer over Mockery so you do not have to manually make complicated Mockery method calls.

Mocking Objects

When mocking an object that is going to be injected into your application via Laravel's service container, you will need to bind your mocked instance into the container as an `instance` binding. This will instruct the container to use your mocked instance of the object instead of constructing the object itself:

```
use App\Service;
use Mockery;
use Mockery\MockInterface;

public function test_something_can_be_mocked()
{
    $this->instance(
        Service::class,
        Mockery::mock(Service::class, function (MockInterface $mock) {
            $mock->shouldReceive('process')->once();
        })
    );
}
```

In order to make this more convenient, you may use the `mock` method that is provided by Laravel's base test case class. For example, the following example is equivalent to the example above:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->mock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

You may use the `partialMock` method when you only need to mock a few methods of an object. The methods that are not mocked will be executed normally when called:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->partialMock(Service::class, function (MockInterface
$mock) {
    $mock->shouldReceive('process')->once();
});
```

Similarly, if you want to spy on an object, Laravel's base test case class offers a `spy` method as a convenient wrapper around the `Mockery::spy` method. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

```
use App\Service;

$spy = $this->spy(Service::class);

// ...

$spy->shouldHaveReceived('process');
```

Mocking Facades

Unlike traditional static method calls, facades (including real-time facades) may be mocked. This provides a great advantage over traditional static methods and grants you the same testability that you would have if you were using traditional dependency injection. When testing, you may often want to mock a call to a Laravel facade that occurs in one of your controllers. For example, consider the following controller action:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Retrieve a list of all users of the application.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

We can mock the call to the `Cache` facade by using the `shouldReceive` method, which will return an instance of a Mockery mock. Since facades are actually resolved and managed by the Laravel service container, they have much more testability than a typical static class. For example, let's mock our call to the `Cache` facade's `get` method:

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

class UserControllerTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

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

        // ...
    }
}
```

{note} You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP testing methods such as `get` and `post` when running your test. Likewise, instead of mocking the `Config` facade, call the `Config::set` method in your tests.

Facade Spies

If you would like to spy on a facade, you may call the `spy` method on the corresponding facade. Spies are similar to mocks; however, spies record any interaction between the spy and the code being tested, allowing you to make assertions after the code is executed:

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

public function test_values_are_be_stored_in_cache()
{
    Cache::spy();

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

    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->once()->with('name', 'Taylor',
10);
}
```

Bus Fake

When testing code that dispatches jobs, you typically want to assert that a given job was dispatched but not actually queue or execute the job. This is because the job's execution can normally be tested in a separate test class.

You may use the `Bus` facade's `fake` method to prevent jobs from being dispatched to the queue. Then, after executing the code under test, you may inspect which jobs the application attempted to dispatch using the `assertDispatched` and `assertNotDispatched` methods:

```
<?php

namespace Tests\Feature;

use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Bus;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Bus::fake();

        // Perform order shipping...

        // Assert that a job was dispatched...
        Bus::assertDispatched(ShipOrder::class);

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);

        // Assert no jobs were dispatched...
        Bus::assertNothingDispatched();
    }
}
```

You may pass a closure to the `assertDispatched` or `assertNotDispatched` methods in

order to assert that a job was dispatched that passes a given "truth test". If at least one job was dispatched that passes the given truth test then the assertion will be successful. For example, you may wish to assert that a job was dispatched for a specific order:

```
Bus::assertDispatched(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

Job Chains

The `Bus` facade's `assertChained` method may be used to assert that a chain of jobs was dispatched. The `assertChained` method accepts an array of chained jobs as its first argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;

Bus::assertChained([
    ShipOrder::class,
    RecordShipment::class,
    UpdateInventory::class
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Bus::assertChained([
    new ShipOrder,
    new RecordShipment,
    new UpdateInventory,
]);
```

Job Batches

The `Bus` facade's `assertBatched` method may be used to assert that a batch of jobs was dispatched. The closure given to the `assertBatched` method receives an instance of `Illuminate\Bus\PendingBatch`, which may be used to inspect the jobs within the batch:

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
           $batch->jobs->count() === 10;
});
```

Event Fake

When testing code that dispatches events, you may wish to instruct Laravel to not actually execute the event's listeners. Using the `Event` facade's `fake` method, you may prevent listeners from executing, execute the code under test, and then assert which events were dispatched by your application using the `assertDispatched`, `assertNotDispatched`, and `assertNothingDispatched` methods:

```php
<?php

namespace Tests\Feature;

use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function test_orders_can_be_shipped()
    {
        Event::fake();

        // Perform order shipping...

        // Assert that an event was dispatched...
        Event::assertDispatched(OrderShipped::class);

        // Assert an event was dispatched twice...
        Event::assertDispatched(OrderShipped::class, 2);

        // Assert an event was not dispatched...
        Event::assertNotDispatched(OrderFailedToShip::class);

        // Assert that no events were dispatched...
        Event::assertNothingDispatched();
    }
}
```

You may pass a closure to the `assertDispatched` or `assertNotDispatched` methods in order to assert that an event was dispatched that passes a given "truth test". If at least one event was dispatched that passes the given truth test then the assertion will be successful:

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

If you would simply like to assert that an event listener is listening to a given event, you may use the `assertListening` method:

```
Event::assertListening(
    OrderShipped::class,
    SendShipmentNotification::class
);
```

{note} After calling `Event::fake()`, no event listeners will be executed. So, if your tests use model factories that rely on events, such as creating a UUID during a model's `creating` event, you should call `Event::fake()` **after** using your factories.

Faking A Subset Of Events

If you only want to fake event listeners for a specific set of events, you may pass them to the `fake` or `fakeFor` method:

```
/**
 * Test order process.
 */
public function test_orders_can_be_processed()
{
    Event::fake([
        OrderCreated::class,
    ]);

    $order = Order::factory()->create();

    Event::assertDispatched(OrderCreated::class);

    // Other events are dispatched as normal...
    $order->update([...]);
}
```

Scoped Event Fakes

If you only want to fake event listeners for a portion of your test, you may use the `fakeFor` method:

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Models\Order;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order process.
     */
    public function test_orders_can_be_processed()
    {
        $order = Event::fakeFor(function () {
            $order = Order::factory()->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });

        // Events are dispatched as normal and observers will run ...
        $order->update([...]);
    }
}
```

HTTP Fake

The `Http` facade's `fake` method allows you to instruct the HTTP client to return stubbed / dummy responses when requests are made. For more information on faking outgoing HTTP requests, please consult the HTTP Client testing documentation.

Mail Fake

You may use the `Mail` facade's `fake` method to prevent mail from being sent. Typically, sending mail is unrelated to the code you are actually testing. Most likely, it is sufficient to simply assert that Laravel was instructed to send a given mailable.

After calling the `Mail` facade's `fake` method, you may then assert that mailables were instructed to be sent to users and even inspect the data the mailables received:

```
<?php

namespace Tests\Feature;

use App\Mail\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Mail;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Mail::fake();

        // Perform order shipping...

        // Assert that no mailables were sent...
        Mail::assertNothingSent();

        // Assert that a mailable was sent...
        Mail::assertSent(OrderShipped::class);

        // Assert a mailable was sent twice...
        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

If you are queueing mailables for delivery in the background, you should use the `assertQueued` method instead of `assertSent`:

```
Mail::assertQueued(OrderShipped::class);

Mail::assertNotQueued(OrderShipped::class);

Mail::assertNothingQueued();
```

You may pass a closure to the `assertSent`, `assertNotSent`, `assertQueued`, or `assertNotQueued` methods in order to assert that a mailable was sent that passes a given "truth test". If at least one mailable was sent that passes the given truth test then the assertion will be successful:

```
Mail::assertSent(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

When calling the `Mail` facade's assertion methods, the mailable instance accepted by the provided closure exposes helpful methods for examining the recipients of the mailable:

```
Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
    return $mail->hasTo($user->email) &&
           $mail->hasCc('...') &&
           $mail->hasBcc('...');
});
```

You may have noticed that there are two methods for asserting that mail was not sent: `assertNotSent` and `assertNotQueued`. Sometimes you may wish to assert that no mail was sent **or** queued. To accomplish this, you may use the `assertNothingOutgoing` and `assertNotOutgoing` methods:

```
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

Notification Fake

You may use the `Notification` facade's `fake` method to prevent notifications from being sent. Typically, sending notifications is unrelated to the code you are actually testing. Most likely, it is sufficient to simply assert that Laravel was instructed to send a given notification.

After calling the `Notification` facade's `fake` method, you may then assert that notifications were instructed to be sent to users and even inspect the data the notifications received:

```php
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Notification::fake();

        // Perform order shipping...

        // Assert that no notifications were sent...
        Notification::assertNothingSent();

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );
    }
}
```

You may pass a closure to the `assertSentTo` or `assertNotSentTo` methods in order to assert that a notification was sent that passes a given "truth test". If at least one notification was sent that passes the given truth test then the assertion will be successful:

```
Notification::assertSentTo(
    $user,
    function (OrderShipped $notification, $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);
```

On-Demand Notifications

If the code you are testing sends on-demand notifications, you will need to assert that the notification was sent to an `Illuminate\Notifications\AnonymousNotifiable` instance:

```
use Illuminate\Notifications\AnonymousNotifiable;

Notification::assertSentTo(
    new AnonymousNotifiable, OrderShipped::class
);
```

By passing a closure as the third argument to the notification assertion methods, you may determine if an on-demand notification was sent to the correct "route" address:

```
Notification::assertSentTo(
    new AnonymousNotifiable,
    OrderShipped::class,
    function ($notification, $channels, $notifiable) use ($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
```

Queue Fake

You may use the `Queue` facade's `fake` method to prevent queued jobs from being pushed to the queue. Most likely, it is sufficient to simply assert that Laravel was instructed to push a given job to the queue since the queued jobs themselves may be tested in another test class.

After calling the `Queue` facade's `fake` method, you may then assert that the application attempted to push jobs to the queue:

```php
<?php

namespace Tests\Feature;

use App\Jobs\AnotherJob;
use App\Jobs\FinalJob;
use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Queue;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped()
    {
        Queue::fake();

        // Perform order shipping...

        // Assert that no jobs were pushed...
        Queue::assertNothingPushed();

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was pushed twice...
        Queue::assertPushed(ShipOrder::class, 2);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::class);
    }
}
```

You may pass a closure to the `assertPushed` or `assertNotPushed` methods in order to assert that a job was pushed that passes a given "truth test". If at least one job was pushed that passes the given truth test then the assertion will be successful:

```php
Queue::assertPushed(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

Job Chains

The `Queue` facade's `assertPushedWithChain` and `assertPushedWithoutChain` methods may be used to inspect the job chain of a pushed job. The `assertPushedWithChain` method accepts the primary job as its first argument and an array of chained jobs as its second argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Queue;

Queue::assertPushedWithChain(ShipOrder::class, [
    RecordShipment::class,
    UpdateInventory::class
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Queue::assertPushedWithChain(ShipOrder::class, [
    new RecordShipment,
    new UpdateInventory,
]);
```

You may use the `assertPushedWithoutChain` method to assert that a job was pushed without a chain of jobs:

```
Queue::assertPushedWithoutChain(ShipOrder::class);
```

Storage Fake

The `Storage` facade's `fake` method allows you to easily generate a fake disk that, combined with the file generation utilities of the `Illuminate\Http\UploadedFile` class, greatly simplifies the testing of file uploads. For example:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_albums_can_be_uploaded()
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);

        // Assert one or more files were stored...
        Storage::disk('photos')->assertExists('photo1.jpg');
        Storage::disk('photos')->assertExists(['photo1.jpg',
'photo2.jpg']);

        // Assert one or more files were not stored...
        Storage::disk('photos')->assertMissing('missing.jpg');
        Storage::disk('photos')->assertMissing(['missing.jpg', 'non-
existing.jpg']);
    }
}
```

For more information on testing file uploads, you may consult the HTTP testing documentation's information on file uploads.

{tip} By default, the `fake` method will delete all files in its temporary directory. If you would like to keep these files, you may use the "persistentFake" method instead.

Interacting With Time

When testing, you may occasionally need to modify the time returned by helpers such as `now` or `Illuminate\Support\Carbon::now()`. Thankfully, Laravel's base feature test class includes helpers that allow you to manipulate the current time:

```
public function testTimeCanBeManipulated()
{
    // Travel into the future...
    $this->travel(5)->milliseconds();
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // Travel into the past...
    $this->travel(-5)->hours();

    // Travel to an explicit time...
    $this->travelTo(now()->subHours(6));

    // Return back to the present time...
    $this->travelBack();
}
```

Notifications

Introduction

In addition to support for sending email, Laravel provides support for sending notifications across a variety of delivery channels, including email, SMS (via Vonage, formerly known as Nexmo), and Slack. In addition, a variety of community built notification channels have been created to send notification over dozens of different channels! Notifications may also be stored in a database so they may be displayed in your web interface.

Typically, notifications should be short, informational messages that notify users of something that occurred in your application. For example, if you are writing a billing application, you might send an "Invoice Paid" notification to your users via the email and SMS channels.

Generating Notifications

In Laravel, each notification is represented by a single class that is typically stored in the `app/Notifications` directory. Don't worry if you don't see this directory in your application - it will be created for you when you run the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid
```

This command will place a fresh notification class in your `app/Notifications` directory. Each notification class contains a `via` method and a variable number of message building methods, such as `toMail` or `toDatabase`, that convert the notification to a message tailored for that particular channel.

Sending Notifications

Using The Notifiable Trait

Notifications may be sent in two ways: using the `notify` method of the `Notifiable` trait or using the `Notification` facade. The `Notifiable` trait is included on your application's `App\Models\User` model by default:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

The `notify` method that is provided by this trait expects to receive a notification instance:

```
use App\Notifications\InvoicePaid;

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

{tip} Remember, you may use the `Notifiable` trait on any of your models. You are not limited to only including it on your `User` model.

Using The Notification Facade

Alternatively, you may send notifications via the `Notification` facade. This approach is useful when you need to send a notification to multiple notifiable entities such as a collection of users. To send notifications using the facade, pass all of the notifiable entities

and the notification instance to the `send` method:

```
use Illuminate\Support\Facades\Notification;

Notification::send($users, new InvoicePaid($invoice));
```

You can also send notifications immediately using the `sendNow` method. This method will send the notification immediately even if the notification implements the `ShouldQueue` interface:

```
Notification::sendNow($developers, new
DeploymentCompleted($deployment));
```

Specifying Delivery Channels

Every notification class has a `via` method that determines on which channels the notification will be delivered. Notifications may be sent on the `mail`, `database`, `broadcast`, `nexmo`, and `slack` channels.

{tip} If you would like to use other delivery channels such as Telegram or Pusher, check out the community driven Laravel Notification Channels website.

The `via` method receives a `$notifiable` instance, which will be an instance of the class to which the notification is being sent. You may use `$notifiable` to determine which channels the notification should be delivered on:

```
/**
 * Get the notification's delivery channels.
 *
 * @param  mixed  $notifiable
 * @return array
 */
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

Queueing Notifications

{note} Before queueing notifications you should configure your queue and start a worker.

Sending notifications can take time, especially if the channel needs to make an external API call to deliver the notification. To speed up your application's response time, let your notification be queued by adding the `ShouldQueue` interface and `Queueable` trait to your class. The interface and trait are already imported for all notifications generated using the `make:notification` command, so you may immediately add them to your notification class:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

Once the `ShouldQueue` interface has been added to your notification, you may send the notification like normal. Laravel will detect the `ShouldQueue` interface on the class and automatically queue the delivery of the notification:

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

If you would like to delay the delivery of the notification, you may chain the `delay` method onto your notification instantiation:

```
$delay = now()->addMinutes(10);

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

You may pass an array to the `delay` method to specify the delay amount for specific

channels:

```
$user->notify((new InvoicePaid($invoice))->delay([
    'mail' => now()->addMinutes(5),
    'sms' => now()->addMinutes(10),
]));
```

When queueing notifications, a queued job will be created for each recipient and channel combination. For example, six jobs will be dispatched to the queue if your notification has three recipients and two channels.

Customizing The Notification Queue Connection

By default, queued notifications will be queued using your application's default queue connection. If you would like to specify a different connection that should be used for a particular notification, you may define a `$connection` property on the notification class:

```
/**
 * The name of the queue connection to use when queueing the
notification.
 *
 * @var string
 */
public $connection = 'redis';
```

Customizing Notification Channel Queues

If you would like to specify a specific queue that should be used for each notification channel supported by the notification, you may define a `viaQueues` method on your notification. This method should return an array of channel name / queue name pairs:

```
/**
 * Determine which queues should be used for each notification channel.
 *
 * @return array
 */
public function viaQueues()
{
    return [
        'mail' => 'mail-queue',
        'slack' => 'slack-queue',
    ];
}
```

Queued Notifications & Database Transactions

When queued notifications are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your notification depends on these models, unexpected errors can occur when the job that sends the queued notification is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued notification should be dispatched after all open database transactions have been committed by calling the `afterCommit` method when sending the notification:

```
use App\Notifications\InvoicePaid;

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

Alternatively, you may call the `afterCommit` method from your notification's constructor:

```php
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

{tip} To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Determining If A Queued Notification Should Be Sent

After a queued notification has been dispatched for the queue for background processing, it will typically be accepted by a queue worker and sent to its intended recipient.

However, if you would like to make the final determination on whether the queued notification should be sent after it is being processed by a queue worker, you may define a `shouldSend` method on the notification class. If this method returns `false`, the notification will not be sent:

```
/**
 * Determine if the notification should be sent.
 *
 * @param  mixed  $notifiable
 * @param  string  $channel
 * @return bool
 */
public function shouldSend($notifiable, $channel)
{
    return $this->invoice->isPaid();
}
```

On-Demand Notifications

Sometimes you may need to send a notification to someone who is not stored as a "user" of your application. Using the `Notification` facade's `route` method, you may specify ad-hoc notification routing information before sending the notification:

```
Notification::route('mail', 'taylor@example.com')
            ->route('nexmo', '5555555555')
            ->route('slack', 'https://hooks.slack.com/services/...')
            ->notify(new InvoicePaid($invoice));
```

If you would like to provide the recipient's name when sending an on-demand notification to the `mail` route, you may provide an array that contains the email address as the key and the name as the value of the first element in the array:

```
Notification::route('mail', [
    'barrett@example.com' => 'Barrett Blair',
])->notify(new InvoicePaid($invoice));
```

Mail Notifications

Formatting Mail Messages

If a notification supports being sent as an email, you should define a `toMail` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\MailMessage` instance.

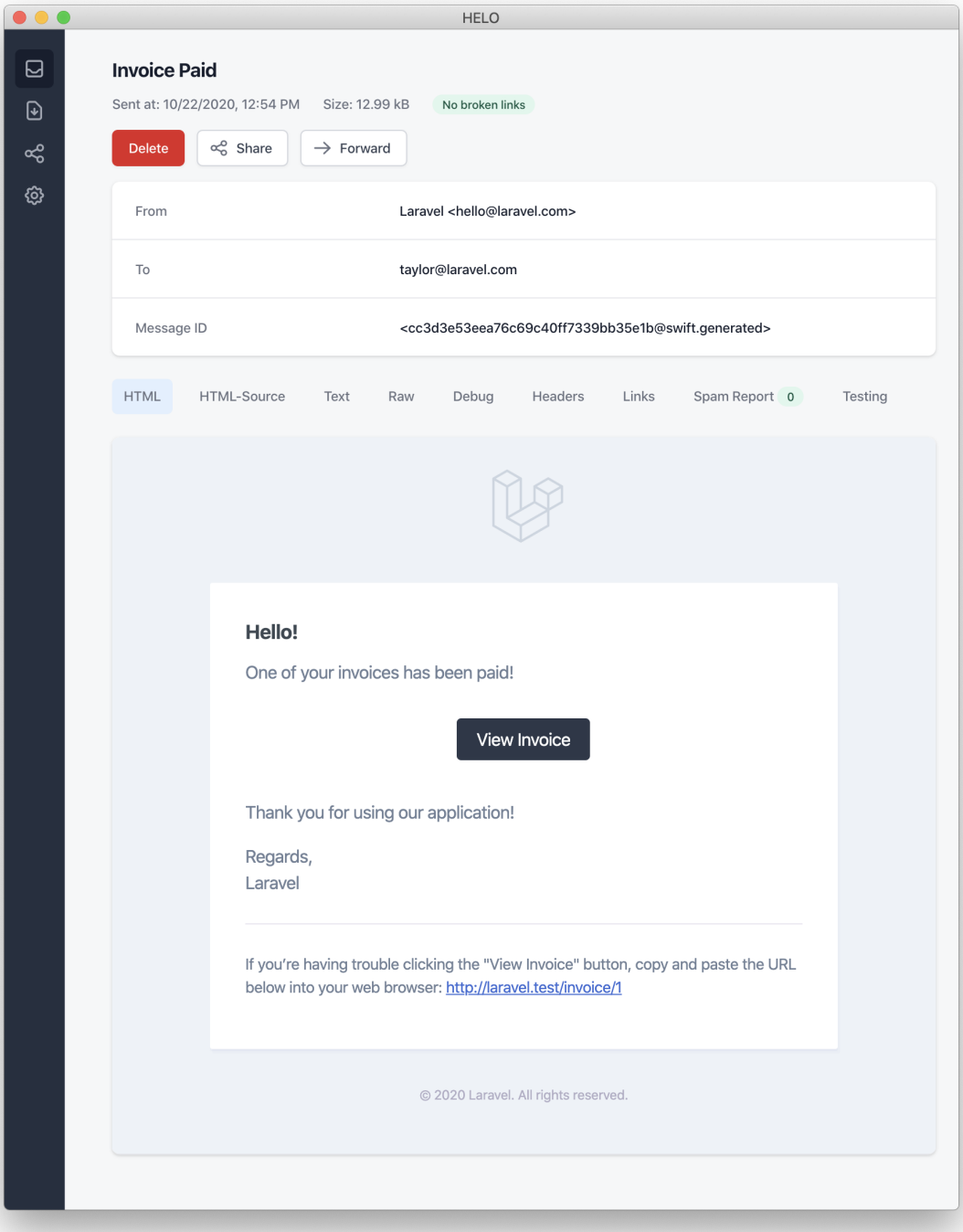
The `MailMessage` class contains a few simple methods to help you build transactional email messages. Mail messages may contain lines of text as well as a "call to action". Let's take a look at an example `toMail` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->greeting('Hello!')
                ->line('One of your invoices has been paid!')
                ->action('View Invoice', $url)
                ->line('Thank you for using our application!');
}
```

{tip} Note we are using `$this->invoice->id` in our `toMail` method. You may pass any data your notification needs to generate its message into the notification's constructor.

In this example, we register a greeting, a line of text, a call to action, and then another line of text. These methods provided by the `MailMessage` object make it simple and fast to format small transactional emails. The mail channel will then translate the message components into a beautiful, responsive HTML email template with a plain-text counterpart. Here is an example of an email generated by the `mail` channel:



{tip} When sending mail notifications, be sure to set the `name` configuration option in your `config/app.php` configuration file. This value will be used in the header and footer of your mail notification messages.

Other Mail Notification Formatting Options

Instead of defining the "lines" of text in the notification class, you may use the `view` method to specify a custom template that should be used to render the notification email:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        'emails.name', ['invoice' => $this->invoice]
    );
}
```

You may specify a plain-text view for the mail message by passing the view name as the second element of an array that is given to the `view` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        ['emails.name.html', 'emails.name.plain'],
        ['invoice' => $this->invoice]
    );
}
```

Error Messages

Some notifications inform users of errors, such as a failed invoice payment. You may indicate that a mail message is regarding an error by calling the `error` method when building your message. When using the `error` method on a mail message, the call to action button will be red instead of black:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Message
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->error()
                ->subject('Notification Subject')
                ->line('...');
}
```

Customizing The Sender

By default, the email's sender / from address is defined in the `config/mail.php` configuration file. However, you may specify the from address for a specific notification using the `from` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->from('barrett@example.com', 'Barrett Blair')
                ->line('...');
}
```

Customizing The Recipient

When sending notifications via the `mail` channel, the notification system will automatically look for an `email` property on your notifiable entity. You may customize which email address is used to deliver the notification by defining a `routeNotificationForMail`

method on the notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return array|string
     */
    public function routeNotificationForMail($notification)
    {
        // Return email address only...
        return $this->email_address;

        // Return email address and name...
        return [$this->email_address => $this->name];
    }
}
```

Customizing The Subject

By default, the email's subject is the class name of the notification formatted to "Title Case". So, if your notification class is named `InvoicePaid`, the email's subject will be `Invoice Paid`. If you would like to specify a different subject for the message, you may call the `subject` method when building your message:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->subject('Notification Subject')
                ->line('...');
}
```

Customizing The Mailer

By default, the email notification will be sent using the default mailer defined in the `config/mail.php` configuration file. However, you may specify a different mailer at runtime by calling the `mailer` method when building your message:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->mailer('postmark')
                ->line('...');
}
```

Customizing The Templates

You can modify the HTML and plain-text template used by mail notifications by publishing the notification package's resources. After running this command, the mail notification templates will be located in the `resources/views/vendor/notifications` directory:

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

Attachments

To add attachments to an email notification, use the `attach` method while building your message. The `attach` method accepts the absolute path to the file as its first argument:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attach('/path/to/file');
}
```

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
                ]);
}
```

Unlike attaching files in mailable objects, you may not attach a file directly from a storage disk using `attachFromStorage`. You should rather use the `attach` method with an absolute path to the file on the storage disk. Alternatively, you could return a mailable from the `toMail` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email)
                ->attachFromStorage('/path/to/file');
}
```

Raw Data Attachments

The `attachData` method may be used to attach a raw string of bytes as an attachment. When calling the `attachData` method, you should provide the filename that should be assigned to the attachment:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->greeting('Hello!')
                ->attachData($this->pdf, 'name.pdf', [
                    'mime' => 'application/pdf',
                ]);
}
```

Using Mailables

If needed, you may return a full mailable object from your notification's `toMail` method. When returning a `Mailable` instead of a `MailMessage`, you will need to specify the message recipient using the mailable object's `to` method:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email);
}
```

Mailables & On-Demand Notifications

If you are sending an on-demand notification, the `$notifiable` instance given to the `toMail` method will be an instance of

`Illuminate\Notifications\AnonymousNotifiable`, which offers a `routeNotificationFor` method that may be used to retrieve the email address the on-demand notification should be sent to:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Notifications\AnonymousNotifiable;

/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    $address = $notifiable instanceof AnonymousNotifiable
            ? $notifiable->routeNotificationFor('mail')
            : $notifiable->email;

    return (new InvoicePaidMailable($this->invoice))
                ->to($address);
}
```

Previewing Mail Notifications

When designing a mail notification template, it is convenient to quickly preview the rendered mail message in your browser like a typical Blade template. For this reason, Laravel allows you to return any mail message generated by a mail notification directly from a route closure or controller. When a `MailMessage` is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
use App\Models\Invoice;
use App\Notifications\InvoicePaid;

Route::get('/notification', function () {
    $invoice = Invoice::find(1);

    return (new InvoicePaid($invoice))
                ->toMail($invoice->user);
});
```

Markdown Mail Notifications

Markdown mail notifications allow you to take advantage of the pre-built templates of mail notifications, while giving you more freedom to write longer, customized messages. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating The Message

To generate a notification with a corresponding Markdown template, you may use the `--markdown` option of the `make:notification` Artisan command:

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

Like all other mail notifications, notifications that use Markdown templates should define a `toMail` method on their notification class. However, instead of using the `line` and `action` methods to construct the notification, use the `markdown` method to specify the name of the Markdown template that should be used. An array of data you wish to make available to the template may be passed as the method's second argument:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->subject('Invoice Paid')
                ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Writing The Message

Markdown mail notifications use a combination of Blade components and Markdown syntax which allow you to easily construct notifications while leveraging Laravel's pre-crafted notification components:

```
@component('mail::message')
# Invoice Paid

Your invoice has been paid!

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

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

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `green`, and `red`. You may add as many button components to a notification as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
View Invoice
@endcomponent
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the notification. This allows you to draw attention to a given block of text:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Customizing The Components

You may export all of the Markdown notification components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

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

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing The CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be in-lined within the HTML representations of your Markdown notifications.
If you would like to build an entirely new theme for Laravel's Markdown components, you

may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of the `mail` configuration file to match the name of your new theme.

To customize the theme for an individual notification, you may call the `theme` method while building the notification's mail message. The `theme` method accepts the name of the theme that should be used when sending the notification:

```
/**
 * Get the mail representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
                ->theme('invoice')
                ->subject('Invoice Paid')
                ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Database Notifications

Prerequisites

The `database` notification channel stores the notification information in a database table. This table will contain information such as the notification type as well as a JSON data structure that describes the notification.

You can query the table to display the notifications in your application's user interface. But, before you can do that, you will need to create a database table to hold your notifications. You may use the `notifications:table` command to generate a migration with the proper table schema:

```
php artisan notifications:table

php artisan migrate
```

Formatting Database Notifications

If a notification supports being stored in a database table, you should define a `toDatabase` or `toArray` method on the notification class. This method will receive a `$notifiable` entity and should return a plain PHP array. The returned array will be encoded as JSON and stored in the `data` column of your `notifications` table. Let's take a look at an example `toArray` method:

```
/**
 * Get the array representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

`toDatabase` Vs. `toArray`

The `toArray` method is also used by the `broadcast` channel to determine which data to broadcast to your JavaScript powered frontend. If you would like to have two different array representations for the `database` and `broadcast` channels, you should define a `toDatabase` method instead of a `toArray` method.

Accessing The Notifications

Once notifications are stored in the database, you need a convenient way to access them from your notifiable entities. The `Illuminate\Notifications\Notifiable` trait, which is included on Laravel's default `App\Models\User` model, includes a `notifications` Eloquent relationship that returns the notifications for the entity. To fetch notifications, you may access this method like any other Eloquent relationship. By default, notifications will be sorted by the `created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

If you want to retrieve only the "unread" notifications, you may use the `unreadNotifications` relationship. Again, these notifications will be sorted by the

`created_at` timestamp with the most recent notifications at the beginning of the collection:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

{tip} To access your notifications from your JavaScript client, you should define a notification controller for your application which returns the notifications for a notifiable entity, such as the current user. You may then make an HTTP request to that controller's URL from your JavaScript client.

Marking Notifications As Read

Typically, you will want to mark a notification as "read" when a user views it. The `Illuminate\Notifications\Notifiable` trait provides a `markAsRead` method, which updates the `read_at` column on the notification's database record:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

However, instead of looping through each notification, you may use the `markAsRead` method directly on a collection of notifications:

```
$user->unreadNotifications->markAsRead();
```

You may also use a mass-update query to mark all of the notifications as read without retrieving them from the database:

```
$user = App\Models\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

You may `delete` the notifications to remove them from the table entirely:

```
$user->notifications()->delete();
```

Broadcast Notifications

Prerequisites

Before broadcasting notifications, you should configure and be familiar with Laravel's event broadcasting services. Event broadcasting provides a way to react to server-side Laravel events from your JavaScript powered frontend.

Formatting Broadcast Notifications

The `broadcast` channel broadcasts notifications using Laravel's event broadcasting services, allowing your JavaScript powered frontend to catch notifications in realtime. If a notification supports broadcasting, you can define a `toBroadcast` method on the notification class. This method will receive a `$notifiable` entity and should return a `BroadcastMessage` instance. If the `toBroadcast` method does not exist, the `toArray` method will be used to gather the data that should be broadcast. The returned data will be encoded as JSON and broadcast to your JavaScript powered frontend. Let's take a look at an example `toBroadcast` method:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the broadcastable representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

Broadcast Queue Configuration

All broadcast notifications are queued for broadcasting. If you would like to configure the queue connection or queue name that is used to queue the broadcast operation, you may use the `onConnection` and `onQueue` methods of the `BroadcastMessage`:

```
return (new BroadcastMessage($data))
                ->onConnection('sqs')
                ->onQueue('broadcasts');
```

Customizing The Notification Type

In addition to the data you specify, all broadcast notifications also have a `type` field containing the full class name of the notification. If you would like to customize the notification `type`, you may define a `broadcastType` method on the notification class:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the type of the notification being broadcast.
 *
 * @return string
 */
public function broadcastType()
{
    return 'broadcast.message';
}
```

Listening For Notifications

Notifications will broadcast on a private channel formatted using a `{notifiable}.{id}` convention. So, if you are sending a notification to an `App\Models\User` instance with an ID of `1`, the notification will be broadcast on the `App.Models.User.1` private channel. When using Laravel Echo, you may easily listen for notifications on a channel using the `notification` method:

```
Echo.private('App.Models.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
```

Customizing The Notification Channel

If you would like to customize which channel that an entity's broadcast notifications are broadcast on, you may define a `receivesBroadcastNotificationsOn` method on the notifiable entity:

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The channels the user receives notification broadcasts on.
     *
     * @return string
     */
    public function receivesBroadcastNotificationsOn()
    {
        return 'users.'.$this->id;
    }
}
```

SMS Notifications

Prerequisites

Sending SMS notifications in Laravel is powered by Vonage (formerly known as Nexmo). Before you can send notifications via Vonage, you need to install the `laravel/nexmo-notification-channel` and `nexmo/laravel` Composer packages

```
composer require laravel/nexmo-notification-channel nexmo/laravel
```

The `nexmo/laravel` package includes its own configuration file. However, you are not required to export this configuration file to your own application. You can simply use the `NEXMO_KEY` and `NEXMO_SECRET` environment variables to set your Vonage public and secret key.

Next, you will need to add a `nexmo` configuration entry to your `config/services.php` configuration file. You may copy the example configuration below to get started:

```
'nexmo' => [
    'sms_from' => '15556666666',
],
```

The `sms_from` option is the phone number that your SMS messages will be sent from. You should generate a phone number for your application in the Vonage control panel.

Formatting SMS Notifications

If a notification supports being sent as an SMS, you should define a `toNexmo` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\NexmoMessage` instance:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your SMS message content');
}
```

Unicode Content

If your SMS message will contain unicode characters, you should call the `unicode` method when constructing the `NexmoMessage` instance:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your unicode message')
                ->unicode();
}
```

Formatting Shortcode Notifications

Laravel also supports sending shortcode notifications, which are pre-defined message templates in your Vonage account. To send a shortcode SMS notification, you should define a `toShortcode` method on your notification class. From within this method, you may return an array specifying the type of notification (`alert`, `2fa`, or `marketing`) as well as the custom values that will populate the template:

```
/**
 * Get the Vonage / Shortcode representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return array
 */
public function toShortcode($notifiable)
{
    return [
        'type' => 'alert',
        'custom' => [
            'code' => 'ABC123',
        ],
    ];
}
```

{tip} Like routing SMS Notifications, you should implement the `routeNotificationForShortcode` method on your notifiable model.

Customizing The "From" Number

If you would like to send some notifications from a phone number that is different from the phone number specified in your `config/services.php` file, you may call the `from` method on a `NexmoMessage` instance:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->content('Your SMS message content')
                ->from('15554443333');
}
```

Adding a Client Reference

If you would like to keep track of costs per user, team, or client, you may add a "client reference" to the notification. Vonage will allow you to generate reports using this client reference so that you can better understand a particular customer's SMS usage. The client reference can be any string up to 40 characters:

```
/**
 * Get the Vonage / SMS representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
                ->clientReference((string) $notifiable->id)
                ->content('Your SMS message content');
}
```

Routing SMS Notifications

To route Vonage notifications to the proper phone number, define a `routeNotificationForNexmo` method on your notifiable entity:

```php
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Nexmo channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return string
     */
    public function routeNotificationForNexmo($notification)
    {
        return $this->phone_number;
    }
}
```

Slack Notifications

Prerequisites

Before you can send notifications via Slack, you must install the Slack notification channel via Composer:

```
composer require laravel/slack-notification-channel
```

You will also need to create a Slack App for your team. After creating the App, you should configure an "Incoming Webhook" for the workspace. Slack will then provide you with a webhook URL that you may use when routing Slack notifications.

Formatting Slack Notifications

If a notification supports being sent as a Slack message, you should define a `toSlack` method on the notification class. This method will receive a `$notifiable` entity and should return an `Illuminate\Notifications\Messages\SlackMessage` instance. Slack messages may contain text content as well as an "attachment" that formats additional text or an array of fields. Let's take a look at a basic `toSlack` example:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->content('One of your invoices has been paid!');
}
```

Customizing The Sender & Recipient

You may use the `from` and `to` methods to customize the sender and recipient. The `from` method accepts a username and emoji identifier, while the `to` method accepts a channel or username:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->from('Ghost', ':ghost:')
                ->to('#bots')
                ->content('This will be sent to #bots');
}
```

You may also use an image as your from "logo" instead of an emoji:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
                ->from('Laravel')
                ->image('https://laravel.com/img/favicon/favicon.ico')
                ->content('This will display the Laravel logo next to
the message');
}
```

Slack Attachments

You may also add "attachments" to Slack messages. Attachments provide richer formatting

options than simple text messages. In this example, we will send an error notification about an exception that occurred in an application, including a link to view more details about the exception:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return \Illuminate\Notifications\Messages\SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
                ->error()
                ->content('Whoops! Something went wrong.')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Exception: File Not Found',
$url)
                               ->content('File [background.jpg] was not
found.');
                });
}
```

Attachments also allow you to specify an array of data that should be presented to the user. The given data will be presented in a table-style format for easy reading:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/invoices/'.$this->invoice->id);

    return (new SlackMessage)
                ->success()
                ->content('One of your invoices has been paid!')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Invoice 1322', $url)
                               ->fields([
                                    'Title' => 'Server Expenses',
                                    'Amount' => '$1,234',
                                    'Via' => 'American Express',
                                    'Was Overdue' => ':-1:',
                                ]);
                });
}
```

Markdown Attachment Content

If some of your attachment fields contain Markdown, you may use the `markdown` method to instruct Slack to parse and display the given attachment fields as Markdown formatted text. The values accepted by this method are: `pretext`, `text`, and / or `fields`. For more information about Slack attachment formatting, check out the Slack API documentation:

```
/**
 * Get the Slack representation of the notification.
 *
 * @param  mixed  $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
                ->error()
                ->content('Whoops! Something went wrong.')
                ->attachment(function ($attachment) use ($url) {
                    $attachment->title('Exception: File Not Found',
$url)
                               ->content('File [background.jpg] was *not
found*.')
                               ->markdown(['text']);
                });
}
```

Routing Slack Notifications

To route Slack notifications to the proper Slack team and channel, define a `routeNotificationForSlack` method on your notifiable entity. This should return the webhook URL to which the notification should be delivered. Webhook URLs may be generated by adding an "Incoming Webhook" service to your Slack team:

```php
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return string
     */
    public function routeNotificationForSlack($notification)
    {
        return 'https://hooks.slack.com/services/...';
    }
}
```

Localizing Notifications

Laravel allows you to send notifications in a locale other than the HTTP request's current locale, and will even remember this locale if the notification is queued.

To accomplish this, the `Illuminate\Notifications\Notification` class offers a `locale` method to set the desired language. The application will change into this locale when the notification is being evaluated and then revert back to the previous locale when evaluation is complete:

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

Localization of multiple notifiable entries may also be achieved via the `Notification` facade:

```
Notification::locale('es')->send(
    $users, new InvoicePaid($invoice)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on your notifiable model, you may instruct Laravel to use this stored locale when sending a notification:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     *
     * @return string
     */
    public function preferredLocale()
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending notifications and mailables to the model. Therefore, there is no need to call the `locale` method when using this interface:

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

Notification Events

Notification Sending Event

When a notification is sending, the `Illuminate\Notifications\Events\NotificationSending` event is dispatched by the notification system. This contains the "notifiable" entity and the notification instance itself. You may register listeners for this event in your application's `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Notifications\Events\NotificationSending' => [
        'App\Listeners\CheckNotificationStatus',
    ],
];
```

The notification will not be sent if an event listener for the `NotificationSending` event returns `false` from its `handle` method:

```
use Illuminate\Notifications\Events\NotificationSending;

/**
 * Handle the event.
 *
 * @param  \Illuminate\Notifications\Events\NotificationSending  $event
 * @return void
 */
public function handle(NotificationSending $event)
{
    return false;
}
```

Within an event listener, you may access the `notifiable`, `notification`, and `channel` properties on the event to learn more about the notification recipient or the notification

itself:

```
/**
 * Handle the event.
 *
 * @param  \Illuminate\Notifications\Events\NotificationSending  $event
 * @return void
 */
public function handle(NotificationSending $event)
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

Notification Sent Event

When a notification is sent, the `Illuminate\Notifications\Events\NotificationSent` event is dispatched by the notification system. This contains the "notifiable" entity and the notification instance itself. You may register listeners for this event in your `EventServiceProvider`:

```
/**
 * 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`, `channel`, and `response` properties on the event to learn more about the notification recipient or the notification itself:

```
/**
 * Handle the event.
 *
 * @param  \Illuminate\Notifications\Events\NotificationSent  $event
 * @return void
 */
public function handle(NotificationSent $event)
{
    // $event->channel
    // $event->notifiable
    // $event->notification
    // $event->response
}
```

Custom Channels

Laravel ships with a handful of notification channels, but you may want to write your own drivers to deliver notifications via other channels. Laravel makes it simple. To get started, define a class that contains a `send` method. The method should receive two arguments: a `$notifiable` and a `$notification`.

Within the `send` method, you may call methods on the notification to retrieve a message object understood by your channel and then send the notification to the `$notifiable` instance however you wish:

```
<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * Send the given notification.
     *
     * @param  mixed  $notifiable
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return void
     */
    public function send($notifiable, Notification $notification)
    {
        $message = $notification->toVoice($notifiable);

        // Send notification to the $notifiable instance...
    }
}
```

Once your notification channel class has been defined, you may return the class name from the `via` method of any of your notifications. In this example, the `toVoice` method of your notification can return whatever object you choose to represent voice messages. For example, you might define your own `VoiceMessage` class to represent these messages:

```php
<?php

namespace App\Notifications;

use App\Channels\Messages\VoiceMessage;
use App\Channels\VoiceChannel;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Get the notification channels.
     *
     * @param  mixed  $notifiable
     * @return array|string
     */
    public function via($notifiable)
    {
        return [VoiceChannel::class];
    }

    /**
     * Get the voice representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return VoiceMessage
     */
    public function toVoice($notifiable)
    {
        // ...
    }
}
```

Laravel Octane

Introduction

Laravel Octane supercharges your application's performance by serving your application using high-powered application servers, including Swoole and RoadRunner. Octane boots your application once, keeps it in memory, and then feeds it requests at supersonic speeds.

Installation

Octane may be installed via the Composer package manager:

```
composer require laravel/octane
```

After installing Octane, you may execute the `octane:install` Artisan command, which will install Octane's configuration file into your application:

```
php artisan octane:install
```

Server Prerequisites

{note} Laravel Octane requires PHP 8.0+.

RoadRunner

RoadRunner is powered by the RoadRunner binary, which is built using Go. The first time you start a RoadRunner based Octane server, Octane will offer to download and install the RoadRunner binary for you.

RoadRunner Via Laravel Sail

If you plan to develop your application using Laravel Sail, you should run the following commands to install Octane and RoadRunner:

```
./vendor/bin/sail up

./vendor/bin/sail composer require laravel/octane spiral/roadrunner
```

Next, you should start a Sail shell and use the `rr` executable to retrieve the latest Linux based build of the RoadRunner binary:

```
./vendor/bin/sail shell

# Within the Sail shell...
./vendor/bin/rr get-binary
```

After installing the RoadRunner binary, you may exit your Sail shell session. You will now need to adjust the `supervisor.conf` file used by Sail to keep your application running. To get started, execute the `sail:publish` Artisan command:

```
./vendor/bin/sail artisan sail:publish
```

Next, update the `command` directive of your application's `docker/supervisord.conf` file

so that Sail serves your application using Octane instead of the PHP development server:

```
command=/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
octane:start --server=roadrunner --host=0.0.0.0 --rpc-port=6001 --
port=8000
```

Finally, ensure the `rr` binary is executable and build your Sail images:

```
chmod +x ./rr

./vendor/bin/sail build --no-cache
```

Swoole

If you plan to use the Swoole application server to serve your Laravel Octane application, you must install the Swoole PHP extension. Typically, this can be done via PECL:

```
pecl install swoole
```

Swoole Via Laravel Sail

{note} Before serving an Octane application via Sail, ensure you have the latest version of Laravel Sail and execute `./vendor/bin/sail build --no-cache` within your application's root directory.

Alternatively, you may develop your Swoole based Octane application using Laravel Sail, the official Docker based development environment for Laravel. Laravel Sail includes the Swoole extension by default. However, you will still need to adjust the `supervisor.conf` file used by Sail to keep your application running. To get started, execute the `sail:publish` Artisan command:

```
./vendor/bin/sail artisan sail:publish
```

Next, update the `command` directive of your application's `docker/supervisord.conf` file so that Sail serves your application using Octane instead of the PHP development server:

```
command=/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan
octane:start --server=swoole --host=0.0.0.0 --port=80
```

Finally, build your Sail images:

```
./vendor/bin/sail build --no-cache
```

Serving Your Application

The Octane server can be started via the `octane:start` Artisan command. By default, this command will utilize the server specified by the `server` configuration option of your application's `octane` configuration file:

```
php artisan octane:start
```

By default, Octane will start the server on port 8000, so you may access your application in a web browser via `http://localhost:8000`.

Serving Your Application Via HTTPS

By default, applications running via Octane generate links prefixed with `http://`. The `OCTANE_HTTPS` environment variable, used within your application's `config/octane.php` configuration file, can be set to `true` when serving your application via HTTPS. When this configuration value is set to `true`, Octane will instruct Laravel to prefix all generated links with `https://`:

```
'https' => env('OCTANE_HTTPS', false),
```

Serving Your Application Via Nginx

{tip} If you aren't quite ready to manage your own server configuration or aren't comfortable configuring all of the various services needed to run a robust Laravel Octane application, check out Laravel Forge.

In production environments, you should serve your Octane application behind a traditional web server such as a Nginx or Apache. Doing so will allow the web server to serve your static assets such as images and stylesheets, as well as manage your SSL certificate termination.

In the Nginx configuration example below file, Nginx will serve the site's static assets and proxy requests to the Octane server that is running on port 8000:

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

server {
    listen 80;
    listen [::]:80;
    server_name domain.com;
    server_tokens off;
    root /home/forge/domain.com/public;

    index index.php;

    charset utf-8;

    location /index.php {
        try_files /not_exists @octane;
    }

    location / {
        try_files $uri $uri/ @octane;
    }

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

    access_log off;
    error_log  /var/log/nginx/domain.com-error.log error;

    error_page 404 /index.php;

    location @octane {
        set $suffix "";

        if ($uri = /index.php) {
            set $suffix ?$query_string;
        }

        proxy_http_version 1.1;
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header SERVER_PORT $server_port;
        proxy_set_header REMOTE_ADDR $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

        proxy_pass http://127.0.0.1:8000$suffix;
    }
}
```

Watching For File Changes

Since your application is loaded in memory once when the Octane server starts, any changes to your application's files will not be reflected when you refresh your browser. For example, route definitions added to your `routes/web.php` file will not be reflected until the server is restarted. For convenience, you may use the `--watch` flag to instruct Octane to automatically restart the server on any file changes within your application:

```
php artisan octane:start --watch
```

Before using this feature, you should ensure that Node is installed within your local development environment. In addition, you should install the Chokidar file-watching library within your project:library:

```
npm install --save-dev chokidar
```

You may configure the directories and files that should be watched using the `watch` configuration option within your application's `config/octane.php` configuration file.

Specifying The Worker Count

By default, Octane will start an application request worker for each CPU core provided by your machine. These workers will then be used to serve incoming HTTP requests as they enter your application. You may manually specify how many workers you would like to start using the `--workers` option when invoking the `octane:start` command:

```
php artisan octane:start --workers=4
```

If you are using the Swoole application server, you may also specify how many "task workers" you wish to start:

```
php artisan octane:start --workers=4 --task-workers=6
```

Specifying The Max Request Count

To help prevent stray memory leaks, Octane can gracefully restart a worker once it has handled a given number of requests. To instruct Octane to do this, you may use the `--max-requests` option:

```
php artisan octane:start --max-requests=250
```

Reloading The Workers

You may gracefully restart the Octane server's application workers using the `octane:reload` command. Typically, this should be done after deployment so that your newly deployed code is loaded into memory and is used to serve to subsequent requests:

```
php artisan octane:reload
```

Stopping The Server

You may stop the Octane server using the `octane:stop` Artisan command:

```
php artisan octane:stop
```

Checking The Server Status

You may check the current status of the Octane server using the `octane:status` Artisan command:

```
php artisan octane:status
```

Dependency Injection & Octane

Since Octane boots your application once and keeps it in memory while serving requests, there are a few caveats you should consider while building your application. For example, the `register` and `boot` methods of your application's service providers will only be executed once when the request worker initially boots. On subsequent requests, the same application instance will be reused.

In light of this, you should take special care when injecting the application service container or request into any object's constructor. By doing so, that object may have a stale version of the container or request on subsequent requests.

Octane will automatically handle resetting any first-party framework state between requests. However, Octane does not always know how to reset the global state created by your application. Therefore, you should be aware of how to build your application in a way that is Octane friendly. Below, we will discuss the most common situations that may cause problems while using Octane.

Container Injection

In general, you should avoid injecting the application service container or HTTP request instance into the constructors of other objects. For example, the following binding injects the entire application service container into an object that is bound as a singleton:

```
use App\Service;

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

In this example, if the `Service` instance is resolved during the application boot process, the

container will be injected into the service and that same container will be held by the `Service` instance on subsequent requests. This **may** not be a problem for your particular application; however, it can lead to the container unexpectedly missing bindings that were added later in the boot cycle or by a subsequent request.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a container resolver closure into the service that always resolves the current container instance:

```
use App\Service;
use Illuminate\Container\Container;

$this->app->bind(Service::class, function ($app) {
    return new Service($app);
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () => Container::getInstance());
});
```

The global `app` helper and the `Container::getInstance()` method will always return the latest version of the application container.

Request Injection

In general, you should avoid injecting the application service container or HTTP request instance into the constructors of other objects. For example, the following binding injects the entire request instance into an object that is bound as a singleton:

```php
use App\Service;

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

In this example, if the `Service` instance is resolved during the application boot process, the HTTP request will be injected into the service and that same request will be held by the `Service` instance on subsequent requests. Therefore, all headers, input, and query string data will be incorrect, as well as all other request data.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a request resolver closure into the service that always resolves the current request instance. Or, the most recommended approach is simply to pass the specific request information your object needs to one of the object's methods at runtime:

```php
use App\Service;

$this->app->bind(Service::class, function ($app) {
    return new Service($app['request']);
});

$this->app->singleton(Service::class, function ($app) {
    return new Service(fn () => $app['request']);
});

// Or...

$service->method($request->input('name'));
```

The global `request` helper will always return the request the application is currently handling and is therefore safe to use within your application.

{note} It is acceptable to type-hint the `Illuminate\Http\Request` instance on your controller methods and route closures.

Configuration Repository Injection

In general, you should avoid injecting the configuration repository instance into the constructors of other objects. For example, the following binding injects the configuration repository into an object that is bound as a singleton:

```
use App\Service;

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

In this example, if the configuration values change between requests, that service will not have access to the new values because it's depending on the original repository instance.

As a work-around, you could either stop registering the binding as a singleton, or you could inject a configuration repository resolver closure to the class:

```
use App\Service;
use Illuminate\Container\Container;

$this->app->bind(Service::class, function ($app) {
    return new Service($app->make('config'));
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () =>
Container::getInstance()->make('config'));
});
```

The global `config` will always return the latest version of the configuration repository and is therefore safe to use within your application.

Managing Memory Leaks

Remember, Octane keeps your application in memory between requests; therefore, adding data to a statically maintained array will result in a memory leak. For example, the following controller has a memory leak since each request to the application will continue to add data to the static `$data` array:

```
use App\Service;
use Illuminate\Http\Request;
use Illuminate\Support\Str;

/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return void
 */
public function index(Request $request)
{
    Service::$data[] = Str::random(10);

    // ...
}
```

While building your application, you should take special care to avoid creating these types of

memory leaks. It is recommended that you monitor your application's memory usage during local development to ensure you are not introducing new memory leaks into your application.

Concurrent Tasks

{note} This feature requires Swoole.

When using Swoole, you may execute operations concurrently via light-weight background tasks. You may accomplish this using Octane's `concurrently` method. You may combine this method with PHP array destructuring to retrieve the results of each operation:

```
use App\User;
use App\Server;
use Laravel\Octane\Facades\Octane;

[$users, $servers] = Octane::concurrently([
    fn () => User::all(),
    fn () => Server::all(),
]);
```

Concurrent tasks processed by Octane utilize Swoole's "task workers", and execute within an entirely different process than the incoming request. The amount of workers available to process concurrent tasks is determined by the `--task-workers` directive on the `octane:start` command:

```
php artisan octane:start --workers=4 --task-workers=6
```

Ticks & Intervals

{note} This feature requires Swoole.

When using Swoole, you may register "tick" operations that will be executed every specified number of seconds. You may register "tick" callbacks via the `tick` method. The first argument provided to the `tick` method should be a string that represents the name of the ticker. The second argument should be a callable that will be invoked at the specified interval.

In this example, we will register a closure to be invoked every 10 seconds. Typically, the `tick` method should be called within the `boot` method of one of your application's service providers:

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
        ->seconds(10);
```

Using the `immediate` method, you may instruct Octane to immediately invoke the tick callback when the Octane server initially boots, and every N seconds thereafter:

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
        ->seconds(10)
        ->immediate();
```

The Octane Cache

{note} This feature requires Swoole.

When using Swoole, you may leverage the Octane cache driver, which provides read and write speeds of up to 2 million operations per second. Therefore, this cache driver is an excellent choice for applications that need extreme read / write speeds from their caching layer.

This cache driver is powered by Swoole tables. All data stored in the cache is available to all workers on the server. However, the cached data will be flushed when the server is restarted:

```
Cache::store('octane')->put('framework', 'Laravel', 30);
```

{tip} The maximum number of entries allowed in the Octane cache may be defined in your application's `octane` configuration file.

Cache Intervals

In addition to the typical methods provided by Laravel's cache system, the Octane cache driver features interval based caches. These caches are automatically refreshed at the specified interval and should be registered within the `boot` method of one of your application's service providers. For example, the following cache will be refreshed every five seconds:

```
use Illuminate\Support\Str;

Cache::store('octane')->interval('random', function () {
    return Str::random(10);
}, seconds: 5)
```

Tables

{note} This feature requires Swoole.

When using Swoole, you may define and interact with your own arbitrary Swoole tables. Swoole tables provide extreme performance throughput and the data in these tables can be accessed by all workers on the server. However, the data within them will be lost when the server is restarted.

Tables should be defined within the `tables` configuration array of your application's `octane` configuration file. An example table that allows a maximum of 1000 rows is already configured for you. The maximum size of string columns may be configured by specifying the column size after the column type as seen below:

```
'tables' => [
    'example:1000' => [
        'name' => 'string:1000',
        'votes' => 'int',
    ],
],
```

To access a table, you may use the `Octane::table` method:

```
use Laravel\Octane\Facades\Octane;

Octane::table('example')->set('uuid', [
    'name' => 'Nuno Maduro',
    'votes' => 1000,
]);

return Octane::table('example')->get('uuid');
```

{note} The column types supported by Swoole tables are: `string`, `int`, and `float`.

Package Development

Introduction

Packages are the primary way of adding functionality to Laravel. Packages might be anything from a great way to work with dates like Carbon or a package that allows you to associate files with Eloquent models like Spatie's Laravel Media Library.

There are different types of packages. Some packages are stand-alone, meaning they work with any PHP framework. Carbon and PHPUnit are examples of stand-alone packages. Any of these packages may be used with Laravel by requiring them in your `composer.json` file.

On the other hand, other packages are specifically intended for use with Laravel. These packages may have routes, controllers, views, and configuration specifically intended to enhance a Laravel application. This guide primarily covers the development of those packages that are Laravel specific.

A Note On Facades

When writing a Laravel application, it generally does not matter if you use contracts or facades since both provide essentially equal levels of testability. However, when writing packages, your package will not typically have access to all of Laravel's testing helpers. If you would like to be able to write your package tests as if the package were installed inside a typical Laravel application, you may use the Orchestral Testbench package.

Package Discovery

In a Laravel application's `config/app.php` configuration file, the `providers` option defines a list of service providers that should be loaded by Laravel. When someone installs your package, you will typically want your service provider to be included in this list. Instead of requiring users to manually add your service provider to the list, you may define the provider in the `extra` section of your package's `composer.json` file. In addition to service providers, you may also list any facades you would like to be registered:

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

Once your package has been configured for discovery, Laravel will automatically register its service providers and facades when it is installed, creating a convenient installation experience for your package's users.

Opting Out Of Package Discovery

If you are the consumer of a package and would like to disable package discovery for a package, you may list the package name in the `extra` section of your application's `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

You may disable package discovery for all packages using the `*` character inside of your application's `dont-discover` directive:

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

Service Providers

Service providers are the connection point between your package and Laravel. A service provider is responsible for binding things into Laravel's service container and informing Laravel where to load package resources such as views, configuration, and localization files.

A service provider extends the `Illuminate\Support\ServiceProvider` class and contains two methods: `register` and `boot`. The base `ServiceProvider` class is located in the `illuminate/support` Composer package, which you should add to your own package's dependencies. To learn more about the structure and purpose of service providers, check out their documentation.

Resources

Configuration

Typically, you will need to publish your package's configuration file to the application's `config` directory. This will allow users of your package to easily override your default configuration options. To allow your configuration files to be published, call the `publishes` method from the `boot` method of your service provider:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/courier.php' => config_path('courier.php'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` command, your file will be copied to the specified publish location. Once your configuration has been published, its values may be accessed like any other configuration file:

```
$value = config('courier.option');
```

{note} You should not define closures in your configuration files. They can not be serialized correctly when users execute the `config:cache` Artisan command.

Default Package Configuration

You may also merge your own package configuration file with the application's published copy. This will allow your users to define only the options they actually want to override in the published copy of the configuration file. To merge the configuration file values, use the `mergeConfigFrom` method within your service provider's `register` method.

The `mergeConfigFrom` method accepts the path to your package's configuration file as its first argument and the name of the application's copy of the configuration file as its second argument:

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```

{note} This method only merges the first level of the configuration array. If your users partially define a multi-dimensional configuration array, the missing options will not be merged.

Routes

If your package contains routes, you may load them using the `loadRoutesFrom` method. This method will automatically determine if the application's routes are cached and will not load your routes file if the routes have already been cached:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/../routes/web.php');
}
```

Migrations

If your package contains database migrations, you may use the `loadMigrationsFrom` method to inform Laravel how to load them. The `loadMigrationsFrom` method accepts the path to your package's migrations as its only argument:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
}
```

Once your package's migrations have been registered, they will automatically be run when the `php artisan migrate` command is executed. You do not need to export them to the application's `database/migrations` directory.

Translations

If your package contains translation files, you may use the `loadTranslationsFrom` method to inform Laravel how to load them. For example, if your package is named `courier`, you should add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/../resources/lang',
'courier');
}
```

Package translations are referenced using the `package::file.line` syntax convention. So, you may load the `courier` package's `welcome` line from the `messages` file like so:

```
echo trans('courier::messages.welcome');
```

Publishing Translations

If you would like to publish your package's translations to the application's `resources/lang/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package paths and their desired publish locations. For example, to publish the translation files for the `courier` package, you may do the following:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/../resources/lang',
'courier');

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

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's translations will be published to the specified publish location.

Views

To register your package's views with Laravel, you need to tell Laravel where the views are located. You may do this using the service provider's `loadViewsFrom` method. The `loadViewsFrom` method accepts two arguments: the path to your view templates and your package's name. For example, if your package's name is `courier`, you would add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

Package views are referenced using the `package::view` syntax convention. So, once your view path is registered in a service provider, you may load the `dashboard` view from the `courier` package like so:

```
Route::get('/dashboard', function () {
    return view('courier::dashboard');
});
```

Overriding Package Views

When you use the `loadViewsFrom` method, Laravel actually registers two locations for your views: the application's `resources/views/vendor` directory and the directory you specify. So, using the `courier` package as an example, Laravel will first check if a custom version of the view has been placed in the `resources/views/vendor/courier` directory by the developer. Then, if the view has not been customized, Laravel will search the package view directory you specified in your call to `loadViewsFrom`. This makes it easy for package users to customize / override your package's views.

Publishing Views

If you would like to make your views available for publishing to the application's `resources/views/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package view paths and their desired publish locations:

```
/**
 * Bootstrap the package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

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

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's views will be copied to the specified publish location.

View Components

If your package contains view components, you may use the `loadViewComponentsAs` method to inform Laravel how to load them. The `loadViewComponentsAs` method accepts two arguments: the tag prefix for your view components and an array of your view component class names. For example, if your package's prefix is `courier` and you have `Alert` and `Button` view components, you would add the following to your service provider's `boot` method:

```
use Courier\Components\Alert;
use Courier\Components\Button;

/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewComponentsAs('courier', [
        Alert::class,
        Button::class,
    ]);
}
```

Once your view components are registered in a service provider, you may reference them in your view like so:

```
<x-courier-alert />

<x-courier-button />
```

Anonymous Components

If your package contains anonymous components, they must be placed within a `components` directory of your package's "views" directory (as specified by `loadViewsFrom`). Then, you may render them by prefixing the component name with the package's view namespace:

```
<x-courier::alert />
```

Commands

To register your package's Artisan commands with Laravel, you may use the `commands` method. This method expects an array of command class names. Once the commands have been registered, you may execute them using the Artisan CLI:

```
use Courier\Console\Commands\InstallCommand;
use Courier\Console\Commands\NetworkCommand;

/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            InstallCommand::class,
            NetworkCommand::class,
        ]);
    }
}
```

Public Assets

Your package may have assets such as JavaScript, CSS, and images. To publish these assets to the application's `public` directory, use the service provider's `publishes` method. In this example, we will also add a `public` asset group tag, which may be used to easily publish groups of related assets:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../public' => public_path('vendor/courier'),
    ], 'public');
}
```

Now, when your package's users execute the `vendor:publish` command, your assets will be copied to the specified publish location. Since users will typically need to overwrite the assets every time the package is updated, you may use the `--force` flag:

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

Publishing File Groups

You may want to publish groups of package assets and resources separately. For instance, you might want to allow your users to publish your package's configuration files without being forced to publish your package's assets. You may do this by "tagging" them when calling the `publishes` method from a package's service provider. For example, let's use tags to define two publish groups for the `courier` package (`courier-config` and `courier-migrations`) in the `boot` method of the package's service provider:

```
/**
 * Bootstrap any package services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'courier-config');

    $this->publishes([
        __DIR__.'/../database/migrations/' =>
database_path('migrations')
    ], 'courier-migrations');
}
```

Now your users may publish these groups separately by referencing their tag when executing the `vendor:publish` command:

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

Laravel Passport

Introduction

Laravel Passport provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the League OAuth2 server that is maintained by Andy Millington and Simon Hamp.

{note} This documentation assumes you are already familiar with OAuth2. If you do not know anything about OAuth2, consider familiarizing yourself with the general terminology and features of OAuth2 before continuing.

Passport Or Sanctum?

Before getting started, you may wish to determine if your application would be better served by Laravel Passport or Laravel Sanctum. If your application absolutely needs to support OAuth2, then you should use Laravel Passport.

However, if you are attempting to authenticate a single-page application, mobile application, or issue API tokens, you should use Laravel Sanctum. Laravel Sanctum does not support OAuth2; however, it provides a much simpler API authentication development experience.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Passport's service provider registers its own database migration directory, so you should migrate your database after installing the package. The Passport migrations will create the tables your application needs to store OAuth2 clients and access tokens:

```
php artisan migrate
```

Next, you should execute the `passport:install` Artisan command. This command will create the encryption keys needed to generate secure access tokens. In addition, the command will create "personal access" and "password grant" clients which will be used to generate access tokens:

```
php artisan passport:install
```

{tip} If you would like to use UUIDs as the primary key value of the Passport `Client` model instead of auto-incrementing integers, please install Passport using the `uuids` option.

After running the `passport:install` command, add the `Laravel\Passport\HasApiTokens` trait to your `App\Models\User` model. This trait will provide a few helper methods to your model which allow you to inspect the authenticated user's token and scopes. If your model is already using the `Laravel\Sanctum\HasApiTokens` trait, you may remove that trait:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

Next, you should call the `Passport::routes` method within the `boot` method of your `App\Providers\AuthServiceProvider`. This method will register the routes necessary to issue access tokens and revoke access tokens, clients, and personal access tokens:

```php
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;
use Illuminate\Support\Facades\Gate;
use Laravel\Passport\Passport;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Models\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        if (! $this->app->routesAreCached()) {
            Passport::routes();
        }
    }
}
```

Finally, in your application's `config/auth.php` configuration file, you should set the `driver` option of the `api` authentication guard to `passport`. This will instruct your application to use Passport's `TokenGuard` when authenticating incoming API requests:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Client UUIDs

You may also run the `passport:install` command with the `--uuids` option present. This option will instruct Passport that you would like to use UUIDs instead of auto-incrementing integers as the Passport `Client` model's primary key values. After running the `passport:install` command with the `--uuids` option, you will be given additional instructions regarding disabling Passport's default migrations:

```
php artisan passport:install --uuids
```

Deploying Passport

When deploying Passport to your application's servers for the first time, you will likely need to run the `passport:keys` command. This command generates the encryption keys Passport needs in order to generate access tokens. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

If necessary, you may define the path where Passport's keys should be loaded from. You may use the `Passport::loadKeysFrom` method to accomplish this. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::loadKeysFrom(__DIR__.'/../secrets/oauth');
}
```

Loading Keys From The Environment

Alternatively, you may publish Passport's configuration file using the `vendor:publish` Artisan command:

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

After the configuration file has been published, you may load your application's encryption keys by defining them as environment variables:

```
PASSPORT_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
<private key here>
-----END RSA PRIVATE KEY-----"

PASSPORT_PUBLIC_KEY="-----BEGIN PUBLIC KEY-----
<public key here>
-----END PUBLIC KEY-----"
```

Migration Customization

If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations

using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-migrations
```

Upgrading Passport

When upgrading to a new major version of Passport, it's important that you carefully review the upgrade guide.

Configuration

Client Secret Hashing

If you would like your client's secrets to be hashed when stored in your database, you should call the `Passport::hashClientSecrets` method in the `boot` method of your `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

Passport::hashClientSecrets();
```

Once enabled, all of your client secrets will only be displayable to the user immediately after they are created. Since the plain-text client secret value is never stored in the database, it is not possible to recover the secret's value if it is lost.

Token Lifetimes

By default, Passport issues long-lived access tokens that expire after one year. If you would like to configure a longer / shorter token lifetime, you may use the `tokensExpireIn`, `refreshTokensExpireIn`, and `personalAccessTokensExpireIn` methods. These methods should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(now()->addDays(15));
    Passport::refreshTokensExpireIn(now()->addDays(30));
    Passport::personalAccessTokensExpireIn(now()->addMonths(6));
}
```

{note} The `expires_at` columns on Passport's database tables are read-only and for display purposes only. When issuing tokens, Passport stores the expiration information within the signed and encrypted tokens. If you need to invalidate a token you should revoke it.

Overriding Default Models

You are free to extend the models used internally by Passport by defining your own model and extending the corresponding Passport model:

```
use Laravel\Passport\Client as PassportClient;

class Client extends PassportClient
{
    // ...
}
```

After defining your model, you may instruct Passport to use your custom model via the `Laravel\Passport\Passport` class. Typically, you should inform Passport about your custom models in the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessClient;
use App\Models\Passport\Token;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::useTokenModel(Token::class);
    Passport::useClientModel(Client::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::usePersonalAccessClientModel(PersonalAccessClient::class);
}
```

Issuing Access Tokens

Using OAuth2 via authorization codes is how most developers are familiar with OAuth2. When using authorization codes, a client application will redirect a user to your server where they will either approve or deny the request to issue an access token to the client.

Managing Clients

First, developers building applications that need to interact with your application's API will need to register their application with yours by creating a "client". Typically, this consists of providing the name of their application and a URL that your application can redirect to after users approve their request for authorization.

The `passport:client` Command

The simplest way to create a client is using the `passport:client` Artisan command. This command may be used to create your own clients for testing your OAuth2 functionality. When you run the `client` command, Passport will prompt you for more information about your client and will provide you with a client ID and secret:

```
php artisan passport:client
```

Redirect URLs

If you would like to allow multiple redirect URLs for your client, you may specify them using a comma-delimited list when prompted for the URL by the `passport:client` command. Any URLs which contain commas should be URL encoded:

```
http://example.com/callback,http://examplefoo.com/callback
```

JSON API

Since your application's users will not be able to utilize the `client` command, Passport provides a JSON API that you may use to create clients. This saves you the trouble of having to manually code controllers for creating, updating, and deleting clients.

However, you will need to pair Passport's JSON API with your own frontend to provide a dashboard for your users to manage their clients. Below, we'll review all of the API endpoints for managing clients. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

`GET /oauth/clients`

This route returns all of the clients for the authenticated user. This is primarily useful for listing all of the user's clients so that they may edit or delete them:

```
axios.get('/oauth/clients')
    .then(response => {
        console.log(response.data);
    });
```

`POST /oauth/clients`

This route is used to create new clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization.

When a client is created, it will be issued a client ID and client secret. These values will be used when requesting access tokens from your application. The client creation route will return the new client instance:

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`PUT /oauth/clients/{client-id}`

This route is used to update clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization. The route will return the updated client instance:

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`DELETE /oauth/clients/{client-id}`

This route is used to delete clients:

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        //
    });
```

Requesting Tokens

Redirecting For Authorization

Once a client has been created, developers may use their client ID and secret to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

{tip} Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Approving The Request

When receiving authorization requests, Passport will automatically display a template to the user allowing them to approve or deny the authorization request. If they approve the request, they will be redirected back to the `redirect_uri` that was specified by the consuming application. The `redirect_uri` must match the `redirect` URL that was specified when the client was created.

If you would like to customize the authorization approval screen, you may publish Passport's views using the `vendor:publish` Artisan command. The published views will be placed in the `resources/views/vendor/passport` directory:

```
php artisan vendor:publish --tag=passport-views
```

Sometimes you may wish to skip the authorization prompt, such as when authorizing a first-party client. You may accomplish this by extending the `Client` model and defining a `skipsAuthorization` method. If `skipsAuthorization` returns `true` the client will be approved and the user will be redirected back to the `redirect_uri` immediately:

```php
<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;

class Client extends BaseClient
{
    /**
     * Determine if the client should skip the authorization prompt.
     *
     * @return bool
     */
    public function skipsAuthorization()
    {
        return $this->firstParty();
    }
}
```

Converting Authorization Codes To Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should first verify the `state` parameter against the value that was stored prior to the redirect. If the state parameter matches then the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request:

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

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

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response =
Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code' => $request->code,
    ]);

    return $response->json();
});
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

{tip} Like the `/oauth/authorize` route, the `/oauth/token` route is defined for you by the `Passport::routes` method. There is no need to manually define this route.

JSON API

Passport also includes a JSON API for managing authorized access tokens. You may pair this with your own frontend to offer your users a dashboard for managing access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints. The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application.

`GET /oauth/tokens`

This route returns all of the authorized access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they can revoke

them:

```
axios.get('/oauth/tokens')
    .then(response => {
        console.log(response.data);
    });
```

`DELETE /oauth/tokens/{token-id}`

This route may be used to revoke authorized access tokens and their related refresh tokens:

```
axios.delete('/oauth/tokens/' + tokenId);
```

Refreshing Tokens

If your application issues short-lived access tokens, users will need to refresh their access tokens via the refresh token that was provided to them when the access token was issued:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token',
[
    'grant_type' => 'refresh_token',
    'refresh_token' => 'the-refresh-token',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => '',
]);

return $response->json();
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

Revoking Tokens

You may revoke a token by using the `revokeAccessToken` method on the `Laravel\Passport\TokenRepository`. You may revoke a token's refresh tokens using the `revokeRefreshTokensByAccessTokenId` method on the `Laravel\Passport\RefreshTokenRepository`. These classes may be resolved using Laravel's service container:

```
use Laravel\Passport\TokenRepository;
use Laravel\Passport\RefreshTokenRepository;

$tokenRepository = app(TokenRepository::class);
$refreshTokenRepository = app(RefreshTokenRepository::class);

// Revoke an access token...
$tokenRepository->revokeAccessToken($tokenId);

// Revoke all of the token's refresh tokens...
$refreshTokenRepository->revokeRefreshTokensByAccessTokenId($tokenId);
```

Purging Tokens

When tokens have been revoked or expired, you might want to purge them from the database. Passport's included `passport:purge` Artisan command can do this for you:

```
# Purge revoked and expired tokens and auth codes...
php artisan passport:purge

# Only purge revoked tokens and auth codes...
php artisan passport:purge --revoked

# Only purge expired tokens and auth codes...
php artisan passport:purge --expired
```

You may also configure a scheduled job in your application's `App\Console\Kernel` class to automatically prune your tokens on a schedule:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('passport:purge')->hourly();
}
```

Authorization Code Grant with PKCE

The Authorization Code grant with "Proof Key for Code Exchange" (PKCE) is a secure way to authenticate single page applications or native applications to access your API. This grant should be used when you can't guarantee that the client secret will be stored confidentially or in order to mitigate the threat of having the authorization code intercepted by an attacker. A combination of a "code verifier" and a "code challenge" replaces the client secret when exchanging the authorization code for an access token.

Creating The Client

Before your application can issue tokens via the authorization code grant with PKCE, you will need to create a PKCE-enabled client. You may do this using the `passport:client` Artisan command with the `--public` option:

```
php artisan passport:client --public
```

Requesting Tokens

Code Verifier & Code Challenge

As this authorization grant does not provide a client secret, developers will need to generate a combination of a code verifier and a code challenge in order to request a token.

The code verifier should be a random string of between 43 and 128 characters containing letters, numbers, and `"-"`, `"."`, `"_"`, `"~"` characters, as defined in the RFC 7636 specification.

The code challenge should be a Base64 encoded string with URL and filename-safe characters. The trailing `'='` characters should be removed and no line breaks, whitespace, or other additional characters should be present.

```
$encoded = base64_encode(hash('sha256', $code_verifier, true));

$codeChallenge = strtr(rtrim($encoded, '='), '+/', '-_');
```

Redirecting For Authorization

Once a client has been created, you may use the client ID and the generated code verifier and code challenge to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route:

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $request->session()->put(
        'code_verifier', $code_verifier = Str::random(128)
    );

    $codeChallenge = strtr(rtrim(
        base64_encode(hash('sha256', $code_verifier, true))
    , '='), '+/', '-_');

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        'code_challenge' => $codeChallenge,
        'code_challenge_method' => 'S256',
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

Converting Authorization Codes To Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming

application. The consumer should verify the `state` parameter against the value that was stored prior to the redirect, as in the standard Authorization Code Grant.

If the state parameter matches, the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request along with the originally generated code verifier:

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

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

    $codeVerifier = $request->session()->pull('code_verifier');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response =
Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code_verifier' => $codeVerifier,
        'code' => $request->code,
    ]);

    return $response->json();
});
```

Password Grant Tokens

The OAuth2 password grant allows your other first-party clients, such as a mobile application, to obtain an access token using an email address / username and password. This allows you to issue access tokens securely to your first-party clients without requiring your users to go through the entire OAuth2 authorization code redirect flow.

Creating A Password Grant Client

Before your application can issue tokens via the password grant, you will need to create a password grant client. You may do this using the `passport:client` Artisan command with the `--password` option. **If you have already run the `passport:install` command, you do not need to run this command:**

```
php artisan passport:client --password
```

Requesting Tokens

Once you have created a password grant client, you may request an access token by issuing a `POST` request to the `/oauth/token` route with the user's email address and password. Remember, this route is already registered by the `Passport::routes` method so there is no need to define it manually. If the request is successful, you will receive an `access_token` and `refresh_token` in the JSON response from the server:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token',
[
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '',
]);

return $response->json();
```

{tip} Remember, access tokens are long-lived by default. However, you are free to configure your maximum access token lifetime if needed.

Requesting All Scopes

When using the password grant or client credentials grant, you may wish to authorize the token for all of the scopes supported by your application. You can do this by requesting the `*` scope. If you request the `*` scope, the `can` method on the token instance will always return `true`. This scope may only be assigned to a token that is issued using the `password` or `client_credentials` grant:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token',
[
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '*',
]);
```

Customizing The User Provider

If your application uses more than one authentication user provider, you may specify which user provider the password grant client uses by providing a `--provider` option when creating the client via the `artisan passport:client --password` command. The given provider name should match a valid provider defined in your application's `config/auth.php` configuration file. You can then protect your route using middleware to ensure that only users from the guard's specified provider are authorized.

Customizing The Username Field

When authenticating using the password grant, Passport will use the `email` attribute of your authenticatable model as the "username". However, you may customize this behavior by defining a `findForPassport` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Find the user instance for the given username.
     *
     * @param  string  $username
     * @return \App\Models\User
     */
    public function findForPassport($username)
    {
        return $this->where('username', $username)->first();
    }
}
```

Customizing The Password Validation

When authenticating using the password grant, Passport will use the `password` attribute of your model to validate the given password. If your model does not have a `password` attribute or you wish to customize the password validation logic, you can define a `validateForPassportPasswordGrant` method on your model:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Hash;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Validate the password of the user for the Passport password
grant.
     *
     * @param  string  $password
     * @return bool
     */
    public function validateForPassportPasswordGrant($password)
    {
        return Hash::check($password, $this->password);
    }
}
```

Implicit Grant Tokens

The implicit grant is similar to the authorization code grant; however, the token is returned to the client without exchanging an authorization code. This grant is most commonly used for JavaScript or mobile applications where the client credentials can't be securely stored. To enable the grant, call the `enableImplicitGrant` method in the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Once the grant has been enabled, developers may use their client ID to request an access token from your application. The consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
use Illuminate\Http\Request;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'token',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

{tip} Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Client Credentials Grant Tokens

The client credentials grant is suitable for machine-to-machine authentication. For example, you might use this grant in a scheduled job which is performing maintenance tasks over an API.

Before your application can issue tokens via the client credentials grant, you will need to create a client credentials grant client. You may do this using the `--client` option of the `passport:client` Artisan command:

```
php artisan passport:client --client
```

Next, to use this grant type, you need to add the `CheckClientCredentials` middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

Then, attach the middleware to a route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client');
```

To restrict access to the route to specific scopes, you may provide a comma-delimited list of the required scopes when attaching the `client` middleware to the route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client:check-status,your-scope');
```

Retrieving Tokens

To retrieve a token using this grant type, make a request to the `oauth/token` endpoint:

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token',
[
    'grant_type' => 'client_credentials',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => 'your-scope',
]);

return $response->json()['access_token'];
```

Personal Access Tokens

Sometimes, your users may want to issue access tokens to themselves without going through the typical authorization code redirect flow. Allowing users to issue tokens to themselves via your application's UI can be useful for allowing users to experiment with your API or may serve as a simpler approach to issuing access tokens in general.

Creating A Personal Access Client

Before your application can issue personal access tokens, you will need to create a personal access client. You may do this by executing the `passport:client` Artisan command with the `--personal` option. If you have already run the `passport:install` command, you do not need to run this command:

```
php artisan passport:client --personal
```

After creating your personal access client, place the client's ID and plain-text secret value in your application's `.env` file:

```
PASSPORT_PERSONAL_ACCESS_CLIENT_ID="client-id-value"
PASSPORT_PERSONAL_ACCESS_CLIENT_SECRET="unhashed-client-secret-value"
```

Managing Personal Access Tokens

Once you have created a personal access client, you may issue tokens for a given user using the `createToken` method on the `App\Models\User` model instance. The `createToken` method accepts the name of the token as its first argument and an optional array of scopes as its second argument:

```
use App\Models\User;

$user = User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport also includes a JSON API for managing personal access tokens. You may pair this with your own frontend to offer your users a dashboard for managing personal access tokens. Below, we'll review all of the API endpoints for managing personal access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

`GET /oauth/scopes`

This route returns all of the scopes defined for your application. You may use this route to list the scopes a user may assign to a personal access token:

```
axios.get('/oauth/scopes')
    .then(response => {
        console.log(response.data);
    });
```

`GET /oauth/personal-access-tokens`

This route returns all of the personal access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they may edit or revoke them:

```
axios.get('/oauth/personal-access-tokens')
    .then(response => {
        console.log(response.data);
    });
```

`POST /oauth/personal-access-tokens`

This route creates new personal access tokens. It requires two pieces of data: the token's `name` and the `scopes` that should be assigned to the token:

```
const data = {
    name: 'Token Name',
    scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
    .then(response => {
        console.log(response.data.accessToken);
    })
    .catch (response => {
        // List errors on response...
    });
```

`DELETE /oauth/personal-access-tokens/{token-id}`

This route may be used to revoke personal access tokens:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protecting Routes

Via Middleware

Passport includes an authentication guard that will validate access tokens on incoming requests. Once you have configured the `api` guard to use the `passport` driver, you only need to specify the `auth:api` middleware on any routes that should require a valid access token:

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

Multiple Authentication Guards

If your application authenticates different types of users that perhaps use entirely different Eloquent models, you will likely need to define a guard configuration for each user provider type in your application. This allows you to protect requests intended for specific user providers. For example, given the following guard configuration the `config/auth.php` configuration file:

```
'api' => [
    'driver' => 'passport',
    'provider' => 'users',
],

'api-customers' => [
    'driver' => 'passport',
    'provider' => 'customers',
],
```

The following route will utilize the `api-customers` guard, which uses the `customers` user provider, to authenticate incoming requests:

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

{tip} For more information on using multiple user providers with Passport, please consult the password grant documentation.

Passing The Access Token

When calling routes that are protected by Passport, your application's API consumers should specify their access token as a `Bearer` token in the `Authorization` header of their request. For example, when using the Guzzle HTTP library:

```
use Illuminate\Support\Facades\Http;

$response = Http::withHeaders([
    'Accept' => 'application/json',
    'Authorization' => 'Bearer '.$accessToken,
])->get('https://passport-app.test/api/user');

return $response->json();
```

Token Scopes

Scopes allow your API clients to request a specific set of permissions when requesting authorization to access an account. For example, if you are building an e-commerce application, not all API consumers will need the ability to place orders. Instead, you may allow the consumers to only request authorization to access order shipment statuses. In other words, scopes allow your application's users to limit the actions a third-party application can perform on their behalf.

Defining Scopes

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your application's `App\Providers\AuthServiceProvider` class. The `tokensCan` method accepts an array of scope names and scope descriptions. The scope description may be anything you wish and will be displayed to users on the authorization approval screen:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

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

Default Scope

If a client does not request any specific scopes, you may configure your Passport server to

attach default scope(s) to the token using the `setDefaultScope` method. Typically, you should call this method from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use Laravel\Passport\Passport;

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

Passport::setDefaultScope([
    'check-status',
    'place-orders',
]);
```

Assigning Scopes To Tokens

When Requesting Authorization Codes

When requesting an access token using the authorization code grant, consumers should specify their desired scopes as the `scope` query string parameter. The `scope` parameter should be a space-delimited list of scopes:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `App\Models\User` model's

`createToken` method, you may pass the array of desired scopes as the second argument to the method:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Checking Scopes

Passport includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given scope. To get started, add the following middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Check For All Scopes

The `scopes` middleware may be assigned to a route to verify that the incoming request's access token has all of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
})->middleware(['auth:api', 'scopes:check-status,place-orders']);
```

Check For Any Scopes

The `scope` middleware may be assigned to a route to verify that the incoming request's access token has *at least one* of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
})->middleware(['auth:api', 'scope:check-status,place-orders']);
```

Checking Scopes On A Token Instance

Once an access token authenticated request has entered your application, you may still check if the token has a given scope using the `tokenCan` method on the authenticated `App\Models\User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        //
    }
});
```

Additional Scope Methods

The `scopeIds` method will return an array of all defined IDs / names:

```
use Laravel\Passport\Passport;

Passport::scopeIds();
```

The `scopes` method will return an array of all defined scopes as instances of `Laravel\Passport\Scope`:

```
Passport::scopes();
```

The `scopesFor` method will return an array of `Laravel\Passport\Scope` instances matching the given IDs / names:

```
Passport::scopesFor(['place-orders', 'check-status']);
```

You may determine if a given scope has been defined using the `hasScope` method:

```
Passport::hasScope('place-orders');
```

Consuming Your API With JavaScript

When building an API, it can be extremely useful to be able to consume your own API from your JavaScript application. This approach to API development allows your own application to consume the same API that you are sharing with the world. The same API may be consumed by your web application, mobile applications, third-party applications, and any SDKs that you may publish on various package managers.

Typically, if you want to consume your API from your JavaScript application, you would need to manually send an access token to the application and pass it with each request to your application. However, Passport includes a middleware that can handle this for you. All you need to do is add the `CreateFreshApiToken` middleware to your `web` middleware group in your `app/Http/Kernel.php` file:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

{note} You should ensure that the `CreateFreshApiToken` middleware is the last middleware listed in your middleware stack.

This middleware will attach a `laravel_token` cookie to your outgoing responses. This cookie contains an encrypted JWT that Passport will use to authenticate API requests from your JavaScript application. The JWT has a lifetime equal to your `session.lifetime` configuration value. Now, since the browser will automatically send the cookie with all subsequent requests, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/api/user')
    .then(response => {
        console.log(response.data);
    });
```

Customizing The Cookie Name

If needed, you can customize the `laravel_token` cookie's name using the `Passport::cookie` method. Typically, this method should be called from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::cookie('custom_name');
}
```

CSRF Protection

When using this method of authentication, you will need to ensure a valid CSRF token header is included in your requests. The default Laravel JavaScript scaffolding includes an Axios instance, which will automatically use the encrypted `XSRF-TOKEN` cookie value to send an `X-XSRF-TOKEN` header on same-origin requests.

{tip} If you choose to send the `X-CSRF-TOKEN` header instead of `X-XSRF-TOKEN`, you will need to use the unencrypted token provided by `csrf_token()`.

Events

Passport raises events when issuing access tokens and refresh tokens. You may use these events to prune or revoke other access tokens in your database. If you would like, you may attach listeners to these events in your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],

    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

Testing

Passport's `actingAs` method may be used to specify the currently authenticated user as well as its scopes. The first argument given to the `actingAs` method is the user instance and the second is an array of scopes that should be granted to the user's token:

```
use App\Models\User;
use Laravel\Passport\Passport;

public function test_servers_can_be_created()
{
    Passport::actingAs(
        User::factory()->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(201);
}
```

Passport's `actingAsClient` method may be used to specify the currently authenticated client as well as its scopes. The first argument given to the `actingAsClient` method is the client instance and the second is an array of scopes that should be granted to the client's token:

```
use Laravel\Passport\Client;
use Laravel\Passport\Passport;

public function test_orders_can_be_retrieved()
{
    Passport::actingAsClient(
        Client::factory()->create(),
        ['check-status']
    );

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

    $response->assertStatus(200);
}
```

Resetting Passwords

Introduction

Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this by hand for every application you create, Laravel provides convenient services for sending password reset links and secure resetting passwords.

{tip} Want to get started fast? Install a Laravel application starter kit in a fresh Laravel application. Laravel's starter kits will take care of scaffolding your entire authentication system, including resetting forgotten passwords.

Model Preparation

Before using the password reset features of Laravel, your application's `App\Models\User` model must use the `Illuminate\Notifications\Notifiable` trait. Typically, this trait is already included on the default `App\Models\User` model that is created with new Laravel applications.

Next, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. The `App\Models\User` model included with the framework already implements this interface, and uses the `Illuminate\Auth\Passwords\CanResetPassword` trait to include the methods needed to implement the interface.

Database Preparation

A table must be created to store your application's password reset tokens. The migration for this table is included in the default Laravel application, so you only need to migrate your database to create this table:

```
php artisan migrate
```

Configuring Trusted Hosts

By default, Laravel will respond to all requests it receives regardless of the content of the HTTP request's `Host` header. In addition, the `Host` header's value will be used when generating absolute URLs to your application during a web request.

Typically, you should configure your web server, such as Nginx or Apache, to only send requests to your application that match a given host name. However, if you do not have the ability to customize your web server directly and need to instruct Laravel to only respond to certain host names, you may do so by enabling the `App\Http\Middleware\TrustHosts` middleware for your application. This is particular important when your application offers password reset functionality.

To learn more about this middleware, please consult the `TrustHosts` middleware documentation.

Routing

To properly implement support for allowing users to reset their passwords, we will need to define several routes. First, we will need a pair of routes to handle allowing the user to request a password reset link via their email address. Second, we will need a pair of routes to handle actually resetting the password once the user visits the password reset link that is emailed to them and completes the password reset form.

Requesting The Password Reset Link

The Password Reset Link Request Form

First, we will define the routes that are needed to request password reset links. To get started, we will define a route that returns a view with the password reset link request form:

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

The view that is returned by this route should have a form containing an `email` field, which will allow the user to request a password reset link for a given email address.

Handling The Form Submission

Next, we will define a route that handles the form submission request from the "forgot password" view. This route will be responsible for validating the email address and sending the password reset request to the corresponding user:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

Route::post('/forgot-password', function (Request $request) {
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
                ? back()->with(['status' => __($status)])
                : back()->withErrors(['email' => __($status)]);
})->middleware('guest')->name('password.email');
```

Before moving on, let's examine this route in more detail. First, the request's `email` attribute is validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to send a password reset link to the user. The password broker will take care of retrieving the user by the given field (in this case, the email address) and sending the user a password reset link via Laravel's built-in notification system.

The `sendResetLink` method returns a "status" slug. This status may be translated using Laravel's localization helpers in order to display a user-friendly message to the user regarding the status of their request. The translation of the password reset status is determined by your application's `resources/lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file.

You may be wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `sendResetLink` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the authentication documentation.

{tip} When manually implementing password resets, you are required to define the contents of the views and routes yourself. If you would like scaffolding that includes all necessary authentication and verification logic, check out the Laravel application starter kits.

Resetting The Password

The Password Reset Form

Next, we will define the routes necessary to actually reset the password once the user clicks on the password reset link that has been emailed to them and provides a new password. First, let's define the route that will display the reset password form that is displayed when the user clicks the reset password link. This route will receive a `token` parameter that we will use later to verify the password reset request:

```
Route::get('/reset-password/{token}', function ($token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

The view that is returned by this route should display a form containing an `email` field, a `password` field, a `password_confirmation` field, and a hidden `token` field, which should contain the value of the secret `$token` received by our route.

Handling The Form Submission

Of course, we need to define a route to actually handle the password reset form submission. This route will be responsible for validating the incoming request and updating the user's password in the database:

```
use Illuminate\Auth\Events\PasswordReset;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;

Route::post('/reset-password', function (Request $request) {
    $request->validate([
        'token' => 'required',
        'email' => 'required|email',
        'password' => 'required|min:8|confirmed',
    ]);

    $status = Password::reset(
        $request->only('email', 'password', 'password_confirmation',
'token'),
        function ($user, $password) {
            $user->forceFill([
                'password' => Hash::make($password)
            ])->setRememberToken(Str::random(60));

            $user->save();

            event(new PasswordReset($user));
        }
    );

    return $status === Password::PASSWORD_RESET
                ? redirect()->route('login')->with('status',
__($status))
                : back()->withErrors(['email' => [__($status)]]);
})->middleware('guest')->name('password.update');
```

Before moving on, let's examine this route in more detail. First, the request's `token`, `email`, and `password` attributes are validated. Next, we will use Laravel's built-in "password broker" (via the `Password` facade) to validate the password reset request credentials.

If the token, email address, and password given to the password broker are valid, the closure passed to the `reset` method will be invoked. Within this closure, which receives the user instance and the plain-text password provided to the password reset form, we may update the user's password in the database.

The `reset` method returns a "status" slug. This status may be translated using Laravel's localization helpers in order to display a user-friendly message to the user regarding the

status of their request. The translation of the password reset status is determined by your application's `resources/lang/{lang}/passwords.php` language file. An entry for each possible value of the status slug is located within the `passwords` language file.

Before moving on, you may be wondering how Laravel knows how to retrieve the user record from your application's database when calling the `Password` facade's `reset` method. The Laravel password broker utilizes your authentication system's "user providers" to retrieve database records. The user provider used by the password broker is configured within the `passwords` configuration array of your `config/auth.php` configuration file. To learn more about writing custom user providers, consult the authentication documentation.

Deleting Expired Tokens

Password reset tokens that have expired will still be present within your database. However, you may easily delete these records using the `auth:clear-resets` Artisan command:

```
php artisan auth:clear-resets
```

If you would like to automate this process, consider adding the command to your application's scheduler:

```
$schedule->command('auth:clear-resets')->everyFifteenMinutes();
```

Customization

Reset Link Customization

You may customize the password reset link URL using the `createUrlUsing` method provided by the `ResetPassword` notification class. This method accepts a closure which receives the user instance that is receiving the notification as well as the password reset link token. Typically, you should call this method from your `App\Providers\AuthServiceProvider` service provider's `boot` method:

```
use Illuminate\Auth\Notifications\ResetPassword;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    ResetPassword::createUrlUsing(function ($user, string $token) {
        return 'https://example.com/reset-password?token='.$token;
    });
}
```

Reset Email Customization

You may easily modify the notification class used to send the password reset link to the user. To get started, override the `sendPasswordResetNotification` method on your `App\Models\User` model. Within this method, you may send the notification using any notification class of your own creation. The password reset `$token` is the first argument received by the method. You may use this `$token` to build the password reset URL of your choice and send your notification to the user:

```
use App\Notifications\ResetPasswordNotification;

/**
 * Send a password reset notification to the user.
 *
 * @param  string  $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $url = 'https://example.com/reset-password?token='.$token;

    $this->notify(new ResetPasswordNotification($url));
}
```

Service Providers

Introduction

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

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

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

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

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

Writing Service Providers

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

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

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

The Register Method

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

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

```php
<?php

namespace App\Providers;

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

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

This service provider only defines a `register` method, and uses that method to define an implementation of `App\Services\Riak\Connection` in the service container. If you're not yet familiar with Laravel's service container, check out its documentation.

The `bindings` And `singletons` Properties

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

```php
<?php

namespace App\Providers;

use App\Contracts\DowntimeNotifier;
use App\Contracts\ServerProvider;
use App\Services\DigitalOceanServerProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

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

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

The Boot Method

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

```php
<?php

namespace App\Providers;

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

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

Boot Method Dependency Injection

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

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

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

Registering Providers

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

To register your provider, add it to the array:

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

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

Deferred Providers

If your provider is **only** registering bindings in the service container, you may choose to defer its registration until one of the registered bindings is actually needed. Deferring the loading of such a provider will improve the performance of your application, since it is not loaded from the filesystem on every request.

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

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

```php
<?php

namespace App\Providers;

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

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

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

Queues

Introduction

While building your web application, you may have some tasks, such as parsing and storing an uploaded CSV file, that take too long to perform during a typical web request. Thankfully, Laravel allows you to easily create queued jobs that may be processed in the background. By moving time intensive tasks to a queue, your application can respond to web requests with blazing speed and provide a better user experience to your customers.

Laravel queues provide a unified queueing API across a variety of different queue backends, such as Amazon SQS, Redis, or even a relational database.

Laravel's queue configuration options are stored in your application's `config/queue.php` configuration file. In this file, you will find connection configurations for each of the queue drivers that are included with the framework, including the database, Amazon SQS, Redis, and Beanstalkd drivers, as well as a synchronous driver that will execute jobs immediately (for use during local development). A `null` queue driver is also included which discards queued jobs.

{tip} Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full Horizon documentation for more information.

Connections Vs. Queues

Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". In your `config/queue.php` configuration file, there is a `connections` configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.

Note that each connection configuration example in the `queue` configuration file contains a `queue` attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the `queue` attribute of the connection configuration:

```
use App\Jobs\ProcessPodcast;

// This job is sent to the default connection's default queue...
ProcessPodcast::dispatch();

// This job is sent to the default connection's "emails" queue...
ProcessPodcast::dispatch()->onQueue('emails');
```

Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a `high` queue, you may run a worker that gives them higher processing priority:

```
php artisan queue:work --queue=high,default
```

Driver Notes & Prerequisites

Database

In order to use the `database` queue driver, you will need a database table to hold the jobs. To generate a migration that creates this table, run the `queue:table` Artisan command. Once the migration has been created, you may migrate your database using the `migrate` command:

```
php artisan queue:table

php artisan migrate
```

Finally, don't forget to instruct your application to use the `database` driver by updating the `QUEUE_CONNECTION` variable in your application's `.env` file:

```
QUEUE_CONNECTION=database
```

Redis

In order to use the `redis` queue driver, you should configure a Redis database connection in your `config/database.php` configuration file.

Redis Cluster

If your Redis queue connection uses a Redis Cluster, your queue names must contain a key hash tag. This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

Blocking

When using the Redis queue, you may use the `block_for` configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.

Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to `5` to indicate that the driver should block for five seconds while waiting for a job to become available:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```

{note} Setting `block_for` to `0` will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as `SIGTERM` from being handled until the next job has been processed.

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

- Amazon SQS: `aws/aws-sdk-php ~3.0` - Beanstalkd: `pda/pheanstalk ~4.0` - Redis: `predis/predis ~1.0` or phpredis PHP extension

Creating Jobs

Generating Job Classes

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. If the `app/Jobs` directory doesn't exist, it will be created when you run the `make:job` Artisan command:

```
php artisan make:job ProcessPodcast
```

The generated class will implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating to Laravel that the job should be pushed onto the queue to run asynchronously.

{tip} Job stubs may be customized using stub publishing.

Class Structure

Job classes are very simple, normally containing only a `handle` method that is invoked when the job is processed by the queue. To get started, let's take a look at an example job class. In this example, we'll pretend we manage a podcast publishing service and need to process the uploaded podcast files before they are published:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
```

```
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The podcast instance.
     *
     * @var \App\Models\Podcast
     */
    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param  App\Models\Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  App\Services\AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }
}
```

In this example, note that we were able to pass an Eloquent model directly into the queued job's constructor. Because of the `SerializesModels` trait that the job is using, Eloquent models and their loaded relationships will be gracefully serialized and unserialized when the job is processing.

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

`handle` Method Dependency Injection

The `handle` method is invoked when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Laravel service container automatically injects these dependencies.

If you would like to take total control over how the container injects dependencies into the `handle` method, you may use the container's `bindMethod` method. The `bindMethod` method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the `handle` method however you wish. Typically, you should call this method from the `boot` method of your `App\Providers\AppServiceProvider` service provider:

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function
($job, $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

{note} Binary data, such as raw image contents, should be passed through the `base64_encode` function before being passed to a queued job. Otherwise, the job may not properly serialize to JSON when being placed on the queue.

Handling Relationships

Because loaded relationships also get serialized, the serialized job string can sometimes become quite large. To prevent relations from being serialized, you can call the `withoutRelations` method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

```
/**
 * Create a new job instance.
 *
 * @param  \App\Models\Podcast  $podcast
 * @return void
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

Unique Jobs

{note} Unique jobs require a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks. In addition, unique job constraints do not apply to jobs within batches.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the `ShouldBeUnique` interface on your job class. This interface does not require you to define any additional methods on your class:

```
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

In the example above, the `UpdateSearchIndex` job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing.

In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

```php
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    /**
     * The product instance.
     *
     * @var \App\Product
     */
    public $product;

    /**
     * The number of seconds after which the job's unique lock will be
released.
     *
     * @var int
     */
    public $uniqueFor = 3600;

    /**
     * The unique ID of the job.
     *
     * @return string
     */
    public function uniqueId()
    {
        return $this->product->id;
    }
}
```

In the example above, the `UpdateSearchIndex` job is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.

Keeping Jobs Unique Until Processing Begins

By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry

attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue,
ShouldBeUniqueUntilProcessing
{
    // ...
}
```

Unique Job Locks

Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a lock with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method that returns the cache driver that should be used:

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

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...

    /**
     * Get the cache driver for the unique job lock.
     *
     * @return \Illuminate\Contracts\Cache\Repository
     */
    public function uniqueVia()
    {
        return Cache::driver('redis');
    }
}
```

{tip} If you only need to limit the concurrent processing of a job, use the `WithoutOverlapping` job middleware instead.

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following `handle` method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

```
use Illuminate\Support\Facades\Redis;

/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function
() {
        info('Lock obtained...');

        // Handle job...
    }, function () {
        // Could not obtain lock...

        return $this->release(5);
    });
}
```

While this code is valid, the implementation of the `handle` method becomes noisy since it is cluttered with Redis rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.

Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in an `app/Jobs/Middleware` directory:

```php
<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * Process the queued job.
     *
     * @param  mixed  $job
     * @param  callable  $next
     * @return mixed
     */
    public function handle($job, $next)
    {
        Redis::throttle('key')
                ->block(0)->allow(1)->every(5)
                ->then(function () use ($job, $next) {
                    // Lock obtained...

                    $next($job);
                }, function () use ($job) {
                    // Could not obtain lock...

                    $job->release(5);
                });
    }
}
```

As you can see, like route middleware, job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

After creating job middleware, they may be attached to a job by returning them from the job's `middleware` method. This method does not exist on jobs scaffolded by the `make:job` Artisan command, so you will need to manually add it to your job class:

```
use App\Jobs\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited];
}
```

{tip} Job middleware can also be assigned to queueable event listeners, mailables, and notifications.

Rate Limiting

Although we just demonstrated how to write your own rate limiting job middleware, Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like route rate limiters, job rate limiters are defined using the `RateLimiter` facade's `for` method.

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a `RateLimiter` in the `boot` method of your `AppServiceProvider`:

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    RateLimiter::for('backups', function ($job) {
        return $job->user->vipCustomer()
                    ? Limit::none()
                    : Limit::perHour(1)->by($job->user->id);
    });
}
```

In the example above, we defined an hourly rate limit; however, you may easily define a rate limit based on minutes using the `perMinute` method. In addition, you may pass any value you wish to the `by` method of the rate limit; however, this value is most often used to segment rate limits by customer:

```
return Limit::perMinute(50)->by($job->user->id);
```

Once you have defined your rate limit, you may attach the rate limiter to your backup job using the `Illuminate\Queue\Middleware\RateLimited` middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited('backups')];
}
```

Releasing a rate limited job back onto the queue will still increment the job's total number of `attempts`. You may wish to tune your `tries` and `maxExceptions` properties on your job class accordingly. Or, you may wish to use the `retryUntil` method to define the amount of time until the job should no longer be attempted.

If you do not want a job to be retried when it is rate limited, you may use the `dontRelease` method:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new RateLimited('backups'))->dontRelease()];
}
```

{tip} If you are using Redis, you may use the `Illuminate\Queue\Middleware\RateLimitedWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

Preventing Job Overlaps

Laravel includes an `Illuminate\Queue\Middleware\WithoutOverlapping` middleware that allows you to prevent job overlaps based on an arbitrary key. This can be helpful when a queued job is modifying a resource that should only be modified by one job

at a time.

For example, let's imagine you have a queued job that updates a user's credit score and you want to prevent credit score update job overlaps for the same user ID. To accomplish this, you can return the `WithoutOverlapping` middleware from your job's `middleware` method:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new WithoutOverlapping($this->user->id)];
}
```

Any overlapping jobs will be released back to the queue. You may also specify the number of seconds that must elapse before the released job will be attempted again:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new
WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the `dontRelease` method:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->dontRelease()];
}
```

The `WithoutOverlapping` middleware is powered by Laravel's atomic lock feature. Sometimes, your job may unexpectedly fail or timeout in such a way that the lock is not released. Therefore, you may explicitly define a lock expiration time using the `expireAfter` method. For example, the example below will instruct Laravel to release the `WithoutOverlapping` lock three minutes after the job has started processing:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new
WithoutOverlapping($this->order->id))->expireAfter(180)];
}
```

{note} The `WithoutOverlapping` middleware requires a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks.

Throttling Exceptions

Laravel includes a `Illuminate\Queue\Middleware\ThrottlesExceptions` middleware that allows you to throttle exceptions. Once the job throws a given number of exceptions, all further attempts to execute the job are delayed until a specified time interval lapses. This middleware is particularly useful for jobs that interact with third-party services that are unstable.

For example, let's imagine a queued job that interacts with a third-party API that begins

throwing exceptions. To throttle exceptions, you can return the `ThrottlesExceptions` middleware from your job's `middleware` method. Typically, this middleware should be paired with a job that implements time based attempts:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new ThrottlesExceptions(10, 5)];
}

/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addMinutes(5);
}
```

The first constructor argument accepted by the middleware is the number of exceptions the job can throw before being throttled, while the second constructor argument is the number of minutes that should elapse before the job is attempted again once it has been throttled. In the code example above, if the job throws 10 exceptions within 5 minutes, we will wait 5 minutes before attempting the job again.

When a job throws an exception but the exception threshold has not yet been reached, the job will typically be retried immediately. However, you may specify the number of minutes such a job should be delayed by calling the `backoff` method when attaching the middleware to the job:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new ThrottlesExceptions(10, 5))->backoff(5)];
}
```

Internally, this middleware uses Laravel's cache system to implement rate limiting, and the job's class name is utilized as the cache "key". You may override this key by calling the by method when attaching the middleware to your job. This may be useful if you have multiple jobs interacting with the same third-party service and you would like them to share a common throttling "bucket":

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new ThrottlesExceptions(10, 10))->by('key')];
}
```

{tip} If you are using Redis, you may use the `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic exception throttling middleware.

Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` method on the job itself. The arguments passed to the `dispatch` method will be given to the job's constructor:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // ...

        ProcessPodcast::dispatch($podcast);
    }
}
```

If you would like to conditionally dispatch a job, you may use the `dispatchIf` and `dispatchUnless` methods:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);

ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

Delayed Dispatching

If you would like to specify that a job should not be immediately available for processing by a queue worker, you may use the `delay` method when dispatching the job. For example, let's specify that a job should not be available for processing until 10 minutes after it has been dispatched:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // ...

        ProcessPodcast::dispatch($podcast)
                    ->delay(now()->addMinutes(10));
    }
}
```

{note} The Amazon SQS queue service has a maximum delay time of 15 minutes.

Dispatching After The Response Is Sent To Browser

Alternatively, the `dispatchAfterResponse` method delays dispatching a job until after the HTTP response is sent to the user's browser. This will still allow the user to begin using the application even though a queued job is still executing. This should typically only be used for jobs that take about a second, such as sending an email. Since they are processed within the current HTTP request, jobs dispatched in this fashion do not require a queue worker to be running in order for them to be processed:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

You may also `dispatch` a closure and chain the `afterResponse` method onto the `dispatch` helper to execute a closure after the HTTP response has been sent to the browser:

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

dispatch(function () {
    Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

Synchronous Dispatching

If you would like to dispatch a job immediately (synchronously), you may use the `dispatchSync` method. When using this method, the job will not be queued and will be executed immediately within the current process:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatchSync($podcast);
    }
}
```

Jobs & Database Transactions

While it is perfectly fine to dispatch jobs within database transactions, you should take special care to ensure that your job will actually be able to execute successfully. When dispatching a job within a transaction, it is possible that the job will be processed by a worker before the transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database.

Thankfully, Laravel provides several methods of working around this problem. First, you may set the `after_commit` connection option in your queue connection's configuration array:

```
'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],
```

When the `after_commit` option is `true`, you may dispatch jobs within database transactions; however, Laravel will wait until all open database transactions have been committed before actually dispatching the job. Of course, if no database transactions are currently open, the job will be dispatched immediately.

If a transaction is rolled back due to an exception that occurs during the transaction, the dispatched jobs that were dispatched during that transaction will be discarded.

{tip} Setting the `after_commit` configuration option to `true` will also cause any queued event listeners, mailables, notifications, and broadcast events to be dispatched after all open database transactions have been committed.

Specifying Commit Dispatch Behavior Inline

If you do not set the `after_commit` queue connection configuration option to `true`, you may still indicate that a specific job should be dispatched after all open database transactions have been committed. To accomplish this, you may chain the `afterCommit` method onto your dispatch operation:

```
use App\Jobs\ProcessPodcast;

ProcessPodcast::dispatch($podcast)->afterCommit();
```

Likewise, if the `after_commit` configuration option is set to `true`, you may indicate that a specific job should be dispatched immediately without waiting for any open database transactions to commit:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

Job Chaining

Job chaining allows you to specify a list of queued jobs that should be run in sequence after the primary job has executed successfully. If one job in the sequence fails, the rest of the jobs will not be run. To execute a queued job chain, you may use the `chain` method provided by the `Bus` facade. Laravel's command bus is a lower level component that queued job dispatching is built on top of:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->dispatch();
```

In addition to chaining job class instances, you may also chain closures:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    function () {
        Podcast::update(...);
    },
])->dispatch();
```

{note} Deleting jobs using the `$this->delete()` method within the job will not prevent chained jobs from being processed. The chain will only stop executing if a job in the chain fails.

Chain Connection & Queue

If you would like to specify the connection and queue that should be used for the chained jobs, you may use the `onConnection` and `onQueue` methods. These methods specify the queue connection and queue name that should be used unless the queued job is explicitly assigned a different connection / queue:

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

Chain Failures

When chaining jobs, you may use the `catch` method to specify a closure that should be invoked if a job within the chain fails. The given callback will receive the `Throwable` instance that caused the job failure:

```
use Illuminate\Support\Facades\Bus;
use Throwable;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->catch(function (Throwable $e) {
    // A job within the chain has failed...
})->dispatch();
```

Customizing The Queue & Connection

Dispatching To A Particular Queue

By pushing jobs to different queues, you may "categorize" your queued jobs and even prioritize how many workers you assign to various queues. Keep in mind, this does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method when dispatching the job:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onQueue('processing');
    }
}
```

Alternatively, you may specify the job's queue by calling the `onQueue` method within the job's constructor:

```php
<?php

namespace App\Jobs;

 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->onQueue('processing');
    }
}
```

Dispatching To A Particular Connection

If your application interacts with multiple queue connections, you may specify which connection to push a job to using the `onConnection` method:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onConnection('sqs');
    }
}
```

You may chain the `onConnection` and `onQueue` methods together to specify the connection and the queue for a job:

```php
ProcessPodcast::dispatch($podcast)
              ->onConnection('sqs')
              ->onQueue('processing');
```

Alternatively, you may specify the job's connection by calling the `onConnection` method within the job's constructor:

```php
<?php

namespace App\Jobs;

 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->onConnection('sqs');
    }
}
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

If one of your queued jobs is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a job may be attempted.

One approach to specifying the maximum number of times a job may be attempted is via the `--tries` switch on the Artisan command line. This will apply to all jobs processed by the worker unless the job being processed specifies a more specific number of times it may be attempted:

```
php artisan queue:work --tries=3
```

If a job exceeds its maximum number of attempts, it will be considered a "failed" job. For more information on handling failed jobs, consult the failed job documentation.

You may take a more granular approach by defining the maximum number of times a job may be attempted on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the `--tries` value provided on the command line:

```
<?php

namespace App\Jobs;

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

Time Based Attempts

As an alternative to defining how many times a job may be attempted before it fails, you may define a time at which the job should no longer be attempted. This allows a job to be attempted any number of times within a given time frame. To define the time at which a job should no longer be attempted, add a `retryUntil` method to your job class. This method should return a `DateTime` instance:

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addMinutes(10);
}
```

{tip} You may also define a `tries` property or `retryUntil` method on your queued event listeners.

Max Exceptions

Sometimes you may wish to specify that a job may be attempted many times, but should fail if the retries are triggered by a given number of unhandled exceptions (as opposed to being released by the `release` method directly). To accomplish this, you may define a `maxExceptions` property on your job class:

```php
<?php

namespace App\Jobs;

use Illuminate\Support\Facades\Redis;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 25;

    /**
     * The maximum number of unhandled exceptions to allow before
failing.
     *
     * @var int
     */
    public $maxExceptions = 3;

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        Redis::throttle('key')->allow(10)->every(60)->then(function () {
            // Lock obtained, process the podcast...
        }, function () {
            // Unable to obtain lock...
            return $this->release(10);
        });
    }
}
```

In this example, the job is released for ten seconds if the application is unable to obtain a Redis lock and will continue to be retried up to 25 times. However, the job will fail if three unhandled exceptions are thrown by the job.

Timeout

{note} The `pcntl` PHP extension must be installed in order to specify job timeouts.

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server.

The maximum number of seconds that jobs can run may be specified using the `--timeout` switch on the Artisan command line:

```
php artisan queue:work --timeout=30
```

If the job exceeds its maximum attempts by continually timing out, it will be marked as failed.

You may also define the maximum number of seconds a job should be allowed to run on the job class itself. If the timeout is specified on the job, it will take precedence over any timeout specified on the command line:

```
<?php

namespace App\Jobs;

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

Sometimes, IO blocking processes such as sockets or outgoing HTTP connections may not respect your specified timeout. Therefore, when using these features, you should always attempt to specify a timeout using their APIs as well. For example, when using Guzzle, you should always specify a connection and request timeout value.

Failing On Timeout

If you would like to indicate that a job should be marked as failed on timeout, you may define the `$failOnTimeout` property on the job class:

```
/**
 * Indicate if the job should be marked as failed on timeout.
 *
 * @var bool
 */
public $failOnTimeout = true;
```

Error Handling

If an exception is thrown while the job is being processed, the job will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The maximum number of attempts is defined by the `--tries` switch used on the `queue:work` Artisan command. Alternatively, the maximum number of attempts may be defined on the job class itself. More information on running the queue worker can be found below.

Manually Releasing A Job

Sometimes you may wish to manually release a job back onto the queue so that it can be attempted again at a later time. You may accomplish this by calling the `release` method:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    // ...

    $this->release();
}
```

By default, the `release` method will release the job back onto the queue for immediate processing. However, by passing an integer to the `release` method you may instruct the queue to not make the job available for processing until a given number of seconds has elapsed:

```
$this->release(10);
```

Manually Failing A Job

Occasionally you may need to manually mark a job as "failed". To do so, you may call the `fail` method:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    // ...

    $this->fail();
}
```

If you would like to mark your job as failed because of an exception that you have caught, you may pass the exception to the `fail` method:

```
$this->fail($exception);
```

{tip} For more information on failed jobs, check out the documentation on dealing with job failures.

Job Batching

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing. Before getting started, you should create a database migration to build a table to contain meta information about your job batches, such as their completion percentage. This migration may be generated using the `queue:batches-table` Artisan command:

```
php artisan queue:batches-table

php artisan migrate
```

Defining Batchable Jobs

To define a batchable job, you should create a queueable job as normal; however, you should add the `Illuminate\Bus\Batchable` trait to the job class. This trait provides access to a `batch` method which may be used to retrieve the current batch that the job is executing within:

```php
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
    use Batchable, Dispatchable, InteractsWithQueue, Queueable,
SerializesModels;

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        if ($this->batch()->cancelled()) {
            // Determine if the batch has been cancelled...

            return;
        }

        // Import a portion of the CSV file...
    }
}
```

Dispatching Batches

To dispatch a batch of jobs, you should use the `batch` method of the `Bus` facade. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the `then`, `catch`, and `finally` methods to define completion callbacks for the batch. Each of these callbacks will receive an `Illuminate\Bus\Batch` instance when they are invoked. In this example, we will imagine we are queueing a batch of jobs that each process a given number of rows from a CSV file:

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

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
    // First batch job failure detected...
})->finally(function (Batch $batch) {
    // The batch has finished executing...
})->dispatch();

return $batch->id;
```

The batch's ID, which may be accessed via the `$batch->id` property, may be used to query the Laravel command bus for information about the batch after it has been dispatched.

{note} Since batch callbacks are serialized and executed at a later time by the Laravel queue, you should not use the `$this` variable within the callbacks.

Naming Batches

Some tools such as Laravel Horizon and Laravel Telescope may provide more user-friendly debug information for batches if batches are named. To assign an arbitrary name to a batch, you may call the `name` method while defining the batch:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import CSV')->dispatch();
```

Batch Connection & Queue

If you would like to specify the connection and queue that should be used for the batched jobs, you may use the `onConnection` and `onQueue` methods. All batched jobs must execute within the same connection and queue:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

Chains Within Batches

You may define a set of chained jobs within a batch by placing the chained jobs within an array. For example, we may execute two job chains in parallel and execute a callback when both job chains have finished processing:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

Adding Jobs To Batches

Sometimes it may be useful to add additional jobs to a batch from within a batched job. This

pattern can be useful when you need to batch thousands of jobs which may take too long to dispatch during a web request. So, instead, you may wish to dispatch an initial batch of "loader" jobs that hydrate the batch with even more jobs:

```
$batch = Bus::batch([
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import Contacts')->dispatch();
```

In this example, we will use the `LoadImportBatch` job to hydrate the batch with additional jobs. To accomplish this, we may use the `add` method on the batch instance that may be accessed via the job's `batch` method:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```

{note} You may only add jobs to a batch from within a job that belongs to the same batch.

Inspecting Batches

The `Illuminate\Bus\Batch` instance that is provided to batch completion callbacks has a variety of properties and methods to assist you in interacting with and inspecting a given batch of jobs:

```
// The UUID of the batch...
$batch->id;

// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;

// The number of jobs that have failed...
$batch->failedJobs;

// The number of jobs that have been processed thus far...
$batch->processedJobs();

// The completion percentage of the batch (0-100)...
$batch->progress();

// Indicates if the batch has finished executing...
$batch->finished();

// Cancel the execution of the batch...
$batch->cancel();

// Indicates if the batch has been cancelled...
$batch->cancelled();
```

Returning Batches From Routes

All `Illuminate\Bus\Batch` instances are JSON serializable, meaning you can return them directly from one of your application's routes to retrieve a JSON payload containing information about the batch, including its completion progress. This makes it convenient to display information about the batch's completion progress in your application's UI.

To retrieve a batch by its ID, you may use the `Bus` facade's `findBatch` method:

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

Route::get('/batch/{batchId}', function (string $batchId) {
    return Bus::findBatch($batchId);
});
```

Cancelling Batches

Sometimes you may need to cancel a given batch's execution. This can be accomplished by calling the `cancel` method on the `Illuminate\Bus\Batch` instance:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

    if ($this->batch()->cancelled()) {
        return;
    }
}
```

As you may have noticed in previous examples, batched jobs should typically check to see if the batch has been cancelled at the beginning of their `handle` method:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    // Continue processing...
}
```

Batch Failures

When a batched job fails, the `catch` callback (if assigned) will be invoked. This callback is only invoked for the first job that fails within the batch.

Allowing Failures

When a job within a batch fails, Laravel will automatically mark the batch as "cancelled". If you wish, you may disable this behavior so that a job failure does not automatically mark the batch as cancelled. This may be accomplished by calling the `allowFailures` method while dispatching the batch:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->allowFailures()->dispatch();
```

Retrying Failed Batch Jobs

For convenience, Laravel provides a `queue:retry-batch` Artisan command that allows you to easily retry all of the failed jobs for a given batch. The `queue:retry-batch` command accepts the UUID of the batch whose failed jobs should be retried:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

Pruning Batches

Without pruning, the `job_batches` table can accumulate records very quickly. To mitigate this, you should schedule the `queue:prune-batches` Artisan command to run daily:

```
$schedule->command('queue:prune-batches')->daily();
```

By default, all finished batches that are more than 24 hours old will be pruned. You may use the `hours` option when calling the command to determine how long to retain batch data. For example, the following command will delete all batches that finished over 48 hours ago:

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

Sometimes, your `jobs_batches` table may accumulate batch records for batches that never completed successfully, such as batches where a job failed and that job was never retried successfully. You may instruct the `queue:prune-batches` command to prune these unfinished batch records using the `unfinished` option:

```
$schedule->command('queue:prune-batches --hours=48 --
unfinished=72')->daily();
```

Queueing Closures

Instead of dispatching a job class to the queue, you may also dispatch a closure. This is great for quick, simple tasks that need to be executed outside of the current request cycle. When dispatching closures to the queue, the closure's code content is cryptographically signed so that it can not be modified in transit:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

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

```
use Throwable;

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

Running The Queue Worker

The `queue:work` Command

Laravel includes an Artisan command that will start a queue worker and process new jobs as they are pushed onto the queue. You may run the worker using the `queue:work` Artisan command. Note that once the `queue:work` command has started, it will continue to run until it is manually stopped or you close your terminal:

```
php artisan queue:work
```

{tip} To keep the `queue:work` process running permanently in the background, you should use a process monitor such as Supervisor to ensure that the queue worker does not stop running.

Remember, queue workers, are long-lived processes and store the booted application state in memory. As a result, they will not notice changes in your code base after they have been started. So, during your deployment process, be sure to restart your queue workers. In addition, remember that any static state created or modified by your application will not be automatically reset between jobs.

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker when you want to reload your updated code or reset the application state; however, this command is significantly less efficient than the `queue:work` command:

```
php artisan queue:listen
```

Running Multiple Queue Workers

To assign multiple workers to a queue and process jobs concurrently, you should simply start multiple `queue:work` processes. This can either be done locally via multiple tabs in your terminal or in production using your process manager's configuration settings. When using Supervisor, you may use the `numprocs` configuration value.

Specifying The Connection & Queue

You may also specify which queue connection the worker should utilize. The connection name passed to the `work` command should correspond to one of the connections defined in your `config/queue.php` configuration file:

```
php artisan queue:work redis
```

By default, the `queue:work` command only processes jobs for the default queue on a given connection. However, you may customize your queue worker even further by only processing particular queues for a given connection. For example, if all of your emails are processed in an `emails` queue on your `redis` queue connection, you may issue the following command to start a worker that only processes that queue:

```
php artisan queue:work redis --queue=emails
```

Processing A Specified Number Of Jobs

The `--once` option may be used to instruct the worker to only process a single job from the queue:

```
php artisan queue:work --once
```

The `--max-jobs` option may be used to instruct the worker to process the given number of jobs and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after processing a given number of jobs, releasing any memory they may have accumulated:

```
php artisan queue:work --max-jobs=1000
```

Processing All Queued Jobs & Then Exiting

The `--stop-when-empty` option may be used to instruct the worker to process all jobs and then exit gracefully. This option can be useful when processing Laravel queues within a Docker container if you wish to shutdown the container after the queue is empty:

```
php artisan queue:work --stop-when-empty
```

Processing Jobs For A Given Number Of Seconds

The `--max-time` option may be used to instruct the worker to process jobs for the given number of seconds and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after processing jobs for a given amount of time, releasing any memory they may have accumulated:

```
// Process jobs for one hour and then exit...
php artisan queue:work --max-time=3600
```

Worker Sleep Duration

When jobs are available on the queue, the worker will keep processing jobs with no delay in between them. However, the `sleep` option determines how many seconds the worker will "sleep" if there are no new jobs available. While sleeping, the worker will not process any new jobs - the jobs will be processed after the worker wakes up again.

```
php artisan queue:work --sleep=3
```

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should release any heavy resources after each job completes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done processing the image.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` configuration file, you may set the default `queue` for your `redis` connection to `low`. However, occasionally you may wish to push a job to a `high` priority queue like so:

```
dispatch((new Job)->onQueue('high'));
```

To start a worker that verifies that all of the `high` queue jobs are processed before continuing to any jobs on the `low` queue, pass a comma-delimited list of queue names to the `work` command:

```
php artisan queue:work --queue=high,low
```

Queue Workers & Deployment

Since queue workers are long-lived processes, they will not notice changes to your code without being restarted. So, the simplest way to deploy an application using queue workers is to restart the workers during your deployment process. You may gracefully restart all of the workers by issuing the `queue:restart` command:

```
php artisan queue:restart
```

This command will instruct all queue workers to gracefully exit after they finish processing their current job so that no existing jobs are lost. Since the queue workers will exit when the `queue:restart` command is executed, you should be running a process manager such as Supervisor to automatically restart the queue workers.

{tip} The queue uses the cache to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Job Expirations & Timeouts

Job Expiration

In your `config/queue.php` configuration file, each queue connection defines a `retry_after` option. This option specifies how many seconds the queue connection should wait before retrying a job that is being processed. For example, if the value of `retry_after` is set to `90`, the job will be released back onto the queue if it has been processing for 90 seconds without being released or deleted. Typically, you should set the `retry_after` value to the maximum number of seconds your jobs should reasonably take

to complete processing.

{note} The only queue connection which does not contain a `retry_after` value is Amazon SQS. SQS will retry the job based on the Default Visibility Timeout which is managed within the AWS console.

Worker Timeouts

The `queue:work` Artisan command exposes a `--timeout` option. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server:

```
php artisan queue:work --timeout=60
```

The `retry_after` configuration option and the `--timeout` CLI option are different, but work together to ensure that jobs are not lost and that jobs are only successfully processed once.

{note} The `--timeout` value should always be at least several seconds shorter than your `retry_after` configuration value. This will ensure that a worker processing a frozen job is always terminated before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

Supervisor Configuration

In production, you need a way to keep your `queue:work` processes running. A `queue:work` process may stop running for a variety of reasons, such as an exceeded worker timeout or the execution of the `queue:restart` command.

For this reason, you need to configure a process monitor that can detect when your `queue:work` processes exit and automatically restart them. In addition, process monitors can allow you to specify how many `queue:work` processes you would like to run concurrently. Supervisor is a process monitor commonly used in Linux environments and we will discuss how to configure it in the following documentation.

Installing Supervisor

Supervisor is a process monitor for the Linux operating system, and will automatically restart your `queue:work` processes if they fail. To install Supervisor on Ubuntu, you may use the following command:

```
sudo apt-get install supervisor
```

{tip} If configuring and managing Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your production Laravel projects.

Configuring Supervisor

Supervisor configuration files are typically stored in the `/etc/supervisor/conf.d` directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a `laravel-worker.conf` file that starts and monitors `queue:work` processes:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --
tries=3 --max-time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

In this example, the `numprocs` directive will instruct Supervisor to run eight `queue:work` processes and monitor all of them, automatically restarting them if they fail. You should change the `command` directive of the configuration to reflect your desired queue connection and worker options.

{note} You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker:*
```

For more information on Supervisor, consult the Supervisor documentation.

Dealing With Failed Jobs

Sometimes your queued jobs will fail. Don't worry, things don't always go as planned! Laravel includes a convenient way to specify the maximum number of times a job should be attempted. After a job has exceeded this number of attempts, it will be inserted into the `failed_jobs` database table. Of course, we will need to create that table if it does not already exist. To create a migration for the `failed_jobs` table, you may use the `queue:failed-table` command:

```
php artisan queue:failed-table

php artisan migrate
```

When running a queue worker process, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:work` command. If you do not specify a value for the `--tries` option, jobs will only be attempted once or as many times as specified by the job class' `$tries` property:

```
php artisan queue:work redis --tries=3
```

Using the `--backoff` option, you may specify how many seconds Laravel should wait before retrying a job that has encountered an exception. By default, a job is immediately released back onto the queue so that it may be attempted again:

```
php artisan queue:work redis --tries=3 --backoff=3
```

If you would like to configure how many seconds Laravel should wait before retrying a job that has encountered an exception on a per-job basis, you may do so by defining a `backoff` property on your job class:

```
/**
 * The number of seconds to wait before retrying the job.
 *
 * @var int
 */
public $backoff = 3;
```

If you require more complex logic for determining the job's backoff time, you may define a `backoff` method on your job class:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return int
*/
public function backoff()
{
    return 3;
}
```

You may easily configure "exponential" backoffs by returning an array of backoff values from the `backoff` method. In this example, the retry delay will be 1 second for the first retry, 5 seconds for the second retry, and 10 seconds for the third retry:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return array
*/
public function backoff()
{
    return [1, 5, 10];
}
```

Cleaning Up After Failed Jobs

When a particular job fails, you may want to send an alert to your users or revert any actions that were partially completed by the job. To accomplish this, you may define a `failed`

method on your job class. The `Throwable` instance that caused the job to fail will be passed to the `failed` method:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The podcast instance.
     *
     * @var \App\Podcast
     */
    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param  \App\Models\Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  \App\Services\AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
```

```
        // Process uploaded podcast...
    }

    /**
     * Handle a job failure.
     *
     * @param  \Throwable  $exception
     * @return void
     */
    public function failed(Throwable $exception)
    {
        // Send user notification of failure, etc...
    }
}
```

{note} A new instance of the job is instantiated before invoking the `failed` method; therefore, any class property modifications that may have occurred within the `handle` method will be lost.

Retrying Failed Jobs

To view all of the failed jobs that have been inserted into your `failed_jobs` database table, you may use the `queue:failed` Artisan command:

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, failure time, and other information about the job. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of `ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece`, issue the following command:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
```

If necessary, you may pass multiple IDs to the command:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
91401d2c-0784-4f43-824c-34f94a33c24d
```

You may also retry all of the failed jobs for a particular queue:

```
php artisan queue:retry --queue=name
```

To retry all of your failed jobs, execute the `queue:retry` command and pass `all` as the ID:

```
php artisan queue:retry all
```

If you would like to delete a failed job, you may use the `queue:forget` command:

```
php artisan queue:forget 91401d2c-0784-4f43-824c-34f94a33c24d
```

{tip} When using Horizon, you should use the `horizon:forget` command to delete a failed job instead of the `queue:forget` command.

To delete all of your failed jobs from the `failed_jobs` table, you may use the `queue:flush` command:

```
php artisan queue:flush
```

Ignoring Missing Models

When injecting an Eloquent model into a job, the model is automatically serialized before being placed on the queue and re-retrieved from the database when the job is processed. However, if the model has been deleted while the job was waiting to be processed by a worker, your job may fail with a `ModelNotFoundException`.

For convenience, you may choose to automatically delete jobs with missing models by setting your job's `deleteWhenMissingModels` property to `true`. When this property is set to `true`, Laravel will quietly discard the job without raising an exception:

```
/**
 * Delete the job if its models no longer exist.
 *
 * @var bool
 */
public $deleteWhenMissingModels = true;
```

Storing Failed Jobs In DynamoDB

Laravel also provides support for storing your failed job records in DynamoDB instead of a relational database table. However, you must create a DynamoDB table to store all of the failed job records. Typically, this table should be named `failed_jobs`, but you should name the table based on the value of the `queue.failed.table` configuration value within your application's `queue` configuration file.

The `failed_jobs` table should have a string primary partition key named `application` and a string primary sort key named `uuid`. The `application` portion of the key will contain your application's name as defined by the `name` configuration value within your application's `app` configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store failed jobs for multiple Laravel applications.

In addition, ensure that you install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

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

Next, set the `queue.failed.driver` configuration option's value to `dynamodb`. In addition, you should define `key`, `secret`, and `region` configuration options within the failed job configuration array. These options will be used to authenticate with AWS. When using the `dynamodb` driver, the `queue.failed.database` configuration option is unnecessary:

```
'failed' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'failed_jobs',
],
```

Pruning Failed Jobs

You may delete all of the records in your application's `failed_jobs` table by invoking the `queue:prune-failed` Artisan command:

```
php artisan queue:prune-failed
```

If you provide the `--hours` option to the command, only the failed job records that were inserted within the last N number of hours will be retained. For example, the following command will delete all of the failed job records that were inserted more than 48 hours ago:

```
php artisan queue:prune-failed --hours=48
```

Failed Job Events

If you would like to register an event listener that will be invoked when a job fails, you may use the `Queue` facade's `failing` method. For example, we may attach a closure to this event from the `boot` method of the `AppServiceProvider` that is included with Laravel:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
}
```

Clearing Jobs From Queues

{tip} When using Horizon, you should use the `horizon:clear` command to clear jobs from the queue instead of the `queue:clear` command.

If you would like to delete all jobs from the default queue of the default connection, you may do so using the `queue:clear` Artisan command:

```
php artisan queue:clear
```

You may also provide the `connection` argument and `queue` option to delete jobs from a specific connection and queue:

```
php artisan queue:clear redis --queue=emails
```

{note} Clearing jobs from queues is only available for the SQS, Redis, and database queue drivers. In addition, the SQS message deletion process takes up to 60 seconds, so jobs sent to the SQS queue up to 60 seconds after you clear the queue might also be deleted.

Monitoring Your Queues

If your queue receives a sudden influx of jobs, it could become overwhelmed, leading to a long wait time for jobs to complete. If you wish, Laravel can alert you when your queue job count exceeds a specified threshold.

To get started, you should schedule the `queue:monitor` command to run every minute. The command accepts the names of the queues you wish to monitor as well as your desired job count threshold:

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

Scheduling this command alone is not enough to trigger a notification alerting you of the queue's overwhelmed status. When the command encounters a queue that has a job count exceeding your threshold, an `Illuminate\Queue\Events\QueueBusy` event will be dispatched. You may listen for this event within your application's `EventServiceProvider` in order to send a notification to you or your development team:

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
    Event::listen(function (QueueBusy $event) {
        Notification::route('mail', 'dev@example.com')
                ->notify(new QueueHasLongWaitTime(
                    $event->connection,
                    $event->queue,
                    $event->size
                ));
    });
}
```

Job Events

Using the `before` and `after` methods on the `Queue` facade, you may specify callbacks to be executed before or after a queued job is processed. These callbacks are a great opportunity to perform additional logging or increment statistics for a dashboard. Typically, you should call these methods from the `boot` method of a service provider. For example, we may use the `AppServiceProvider` that is included with Laravel:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }
}
```

Using the `looping` method on the `Queue` facade, you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a

closure to rollback any transactions that were left open by a previously failed job:

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

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

Rate Limiting

Introduction

Laravel includes a simple to use rate limiting abstraction which, in conjunction with your application's cache, provides an easy way to limit any action during a specified window of time.

{tip} If you are interested in rate limiting incoming HTTP requests, please consult the rate limiter middleware documentation.

Cache Configuration

Typically, the rate limiter utilizes your default application cache as defined by the `default` key within your application's `cache` configuration file. However, you may specify which cache driver the rate limiter should use by defining a `limiter` key within your application's `cache` configuration file:

```
'default' => 'memcached',

'limiter' => 'redis',
```

Basic Usage

The `Illuminate\Support\Facades\RateLimiter` facade may be used to interact with the rate limiter. The simplest method offered by the rate limiter is the `attempt` method, which rate limits a given callback for a given number of seconds.

The `attempt` method returns `false` when the callback has no remaining attempts available; otherwise, the `attempt` method will return the callback's result or `true`. The first argument accepted by the `attempt` method is a rate limiter "key", which may be any string of your choosing that represents the action being rate limited:

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

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

if (! $executed) {
  return 'Too many messages sent!';
}
```

Manually Incrementing Attempts

If you would like to manually interact with the rate limiter, a variety of other methods are available. For example, you may invoke the `tooManyAttempts` method to determine if a given rate limiter key has exceeded its maximum number of allowed attempts per minute:

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

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

Alternatively, you may use the `remaining` method to retrieve the number of attempts remaining for a given key. If a given key has retries remaining, you may invoke the `hit` method to increment the number of total attempts:

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

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

    // Send message...
}
```

Determining Limiter Availability

When a key has no more attempts left, the `availableIn` method returns the number of seconds remaining until more attempts will be available:

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

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

    return 'You may try again in '.$seconds.' seconds.';
}
```

Clearing Attempts

You may reset the number of attempts for a given rate limiter key using the `clear` method. For example, you may reset the number of attempts when a given message is read by the

receiver:

```
use App\Models\Message;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Mark the message as read.
 *
 * @param  \App\Models\Message  $message
 * @return \App\Models\Message
 */
public function read(Message $message)
{
    $message->markAsRead();
    RateLimiter::clear('send-message:'.$message->user_id);
    return $message;
}
```

Laravel Documentation

You can find the online version of the Laravel documentation at https://laravel.com/docs

Contribution Guidelines

If you are submitting documentation for the **current stable release**, submit it to the corresponding branch. For example, documentation for Laravel 8 would be submitted to the `8.x` branch. Documentation intended for the next release of Laravel should be submitted to the `master` branch.

HTTP Redirects

Creating 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 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:

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

return redirect()->action([HomeController::class, 'index']);
```

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

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

Redirecting 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!');
});
```

You may use the `withInput` method provided by the `RedirectResponse` instance to flash the current request's input data to the session before redirecting the user to a new location. Once the input has been flashed to the session, you may easily retrieve it during the next request:

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

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
```

Redis

Introduction

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Before using Redis with Laravel, we encourage you to install and use the phpredis PHP extension via PECL. The extension is more complex to install compared to "user-land" PHP packages but may yield better performance for applications that make heavy use of Redis. If you are using Laravel Sail, this extension is already installed in your application's Docker container.

If you are unable to install the phpredis extension, you may install the `predis/predis` package via Composer. Predis is a Redis client written entirely in PHP and does not require any additional extensions:

```
composer require predis/predis
```

Configuration

You may configure your application's Redis settings via the `config/database.php` configuration file. Within this file, you will see a `redis` array containing the Redis servers utilized by your application:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

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

    'cache' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_CACHE_DB', 1),
    ],

],
```

Each Redis server defined in your configuration file is required to have a name, host, and a port unless you define a single URL to represent the Redis connection:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'url' => 'tcp://127.0.0.1:6379?database=0',
    ],

    'cache' => [
        'url' => 'tls://user:password@127.0.0.1:6380?database=1',
    ],

],
```

Configuring The Connection Scheme

By default, Redis clients will use the `tcp` scheme when connecting to your Redis servers; however, you may use TLS / SSL encryption by specifying a `scheme` configuration option in your Redis server's configuration array:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

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

],
```

Clusters

If your application is utilizing a cluster of Redis servers, you should define these clusters within a `clusters` key of your Redis configuration. This configuration key does not exist by default so you will need to create it within your application's `config/database.php`

configuration file:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

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

],
```

By default, clusters will perform client-side sharding across your nodes, allowing you to pool nodes and create a large amount of available RAM. However, client-side sharding does not handle failover; therefore, it is primarily suited for transient cached data that is available from another primary data store.

If you would like to use native Redis clustering instead of client-side sharding, you may specify this by setting the `options.cluster` configuration value to `redis` within your application's `config/database.php` configuration file:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],

    'clusters' => [
        // ...
    ],

],
```

Predis

If you would like your application to interact with Redis via the Predis package, you should ensure the `REDIS_CLIENT` environment variable's value is `predis`:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'predis'),

    // Rest of Redis configuration...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, Predis supports additional connection parameters that may be defined for each of your Redis servers. To utilize these additional configuration options, add them to your Redis server configuration in your application's `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

The Redis Facade Alias

Laravel's `config/app.php` configuration file contains an `aliases` array which defines all of the class aliases that will be registered by the framework. For convenience, an alias entry is included for each facade offered by Laravel; however, the `Redis` alias is disabled because it conflicts with the `Redis` class name provided by the phpredis extension. If you are using the Predis client and would like to enable this alias, you may un-comment the alias in your application's `config/app.php` configuration file.

phpredis

By default, Laravel will use the phpredis extension to communicate with Redis. The client that Laravel will use to communicate with Redis is dictated by the value of the

`redis.client` configuration option, which typically reflects the value of the `REDIS_CLIENT` environment variable:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    // Rest of Redis configuration...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, phpredis supports the following additional connection parameters: `name`, `persistent`, `prefix`, `read_timeout`, `retry_interval`, `timeout`, and `context`. You may add any of these options to your Redis server configuration in the `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
    'context' => [
        // 'auth' => ['username', 'secret'],
        // 'stream' => ['verify_peer' => false],
    ],
],
```

Interacting With Redis

You may interact with Redis by calling various methods on the `Redis` facade. The `Redis` facade supports dynamic methods, meaning you may call any Redis command on the facade and the command will be passed directly to Redis. In this example, we will call the Redis `GET` command by calling the `get` method on the `Redis` facade:

```
<?php

namespace App\Http\Controllers;

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

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

As mentioned above, you may call any of Redis' commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server. If a Redis command expects arguments, you should pass those to the facade's corresponding method:

```
use Illuminate\Support\Facades\Redis;

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may pass commands to the server using the `Redis` facade's `command` method, which accepts the name of the command as its first argument and an array of values as its second argument:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Using Multiple Redis Connections

Your application's `config/database.php` configuration file allows you to define multiple Redis connections / servers. You may obtain a connection to a specific Redis connection using the `Redis` facade's `connection` method:

```
$redis = Redis::connection('connection-name');
```

To obtain an instance of the default Redis connection, you may call the `connection` method without any additional arguments:

```
$redis = Redis::connection();
```

Transactions

The `Redis` facade's `transaction` method provides a convenient wrapper around Redis' native `MULTI` and `EXEC` commands. The `transaction` method accepts a closure as its only argument. This closure will receive a Redis connection instance and may issue any commands it would like to this instance. All of the Redis commands issued within the closure will be executed in a single, atomic transaction:

```
use Illuminate\Support\Facades\Redis;

Redis::transaction(function ($redis) {
    $redis->incr('user_visits', 1);
    $redis->incr('total_visits', 1);
});
```

{note} When defining a Redis transaction, you may not retrieve any values from the Redis connection. Remember, your transaction is executed as a single, atomic operation and that operation is not executed until your entire closure has finished executing its commands.

Lua Scripts

The `eval` method provides another method of executing multiple Redis commands in a single, atomic operation. However, the `eval` method has the benefit of being able to interact with and inspect Redis key values during that operation. Redis scripts are written in the Lua programming language.

The `eval` method can be a bit scary at first, but we'll explore a basic example to break the ice. The `eval` method expects several arguments. First, you should pass the Lua script (as a string) to the method. Secondly, you should pass the number of keys (as an integer) that the script interacts with. Thirdly, you should pass the names of those keys. Finally, you may pass any other additional arguments that you need to access within your script.

In this example, we will increment a counter, inspect its new value, and increment a second counter if the first counter's value is greater than five. Finally, we will return the value of the first counter:

```
$value = Redis::eval(<<<'LUA'
    local counter = redis.call("incr", KEYS[1])

    if counter > 5 then
        redis.call("incr", KEYS[2])
    end

    return counter
LUA, 2, 'first-counter', 'second-counter');
```

{note} Please consult the Redis documentation for more information on Redis scripting.

Pipelining Commands

Sometimes you may need to execute dozens of Redis commands. Instead of making a network trip to your Redis server for each command, you may use the `pipeline` method. The `pipeline` method accepts one argument: a closure that receives a Redis instance. You may issue all of your commands to this Redis instance and they will all be sent to the Redis server at the same time to reduce network trips to the server. The commands will still be executed in the order they were issued:

```
use Illuminate\Support\Facades\Redis;

Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel provides a convenient interface to the Redis `publish` and `subscribe` commands. These Redis commands allow you to listen for messages on a given "channel". You may publish messages to the channel from another application, or even using another programming language, allowing easy communication between applications and processes.

First, let's setup a channel listener using the `subscribe` method. We'll place this method call within an Artisan command since calling the `subscribe` method begins a long-running process:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function ($message) {
            echo $message;
        });
    }
}
```

Now we may publish messages to the channel using the `publish` method:

```
use Illuminate\Support\Facades\Redis;

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

    Redis::publish('test-channel', json_encode([
        'name' => 'Adam Wathan'
    ]));
});
```

Wildcard Subscriptions

Using the `psubscribe` method, you may subscribe to a wildcard channel, which may be useful for catching all messages on all channels. The channel name will be passed as the second argument to the provided closure:

```
Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {
    echo $message;
});
```

Release Notes

Versioning Scheme

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

When referencing the Laravel framework or its components from your application or package, you should always use a version constraint such as `^8.0`, since major releases of Laravel do include breaking changes. However, we strive to always ensure you may update to a new major release in one day or less.

Exceptions

Named Arguments

At this time, PHP's named arguments functionality are not covered by Laravel's backwards compatibility guidelines. We may choose to rename function parameters when necessary in order to improve the Laravel codebase. Therefore, using named arguments when calling Laravel methods should be done cautiously and with the understanding that the parameter names may change in the future.

Support Policy

For LTS releases, such as Laravel 9, 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 18 months and security fixes are provided for 2 years. For all additional libraries, including Lumen, only the latest release receives bug fixes. In addition, please review the database versions supported by Laravel.

Version	PHP (*)	Release	Bug Fixes Until	Security Fixes Until
6 (LTS)	7.2 - 8.0	September 3rd, 2019	January 25th, 2022	September 6th, 2022
7	7.2 - 8.0	March 3rd, 2020	October 6th, 2020	March 3rd, 2021
8	7.3 - 8.1	September 8th, 2020	July 26th, 2022	January 24th, 2023
9 (LTS)	8.0 - 8.1	January 25th, 2022	January 30th, 2024	January 28th, 2025
10	8.0 - 8.1	January 24th, 2023	July 30th, 2024	January 28th, 2025

End of life
Security fixes only

(*) Supported PHP versions

Laravel 8

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

Laravel Jetstream

Laravel Jetstream was written by Taylor Otwell.

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

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

Models Directory

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

Model Factory Classes

Model factory classes were contributed by Taylor Otwell.

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

```php
<?php

namespace Database\Factories;

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

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

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

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

```
use App\Models\User;

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

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

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

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

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

```
use App\Models\User;

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

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

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

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

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

Migration Squashing

Migration squashing was contributed by Taylor Otwell.

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

```
php artisan schema:dump

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

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

Job Batching

Job batching was contributed by Taylor Otwell & Mohamed Said.

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

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

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

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

return $batch->id;
```

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

Improved Rate Limiting

Rate limiting improvements were contributed by Taylor Otwell.

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

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

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

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

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

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

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

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

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

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

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

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

Improved Maintenance Mode

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

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

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

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

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

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

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

Pre-Rendering The Maintenance Mode View

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

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

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

Closure Dispatch / Chain `catch`

Catch improvements were contributed by Mohamed Said.

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

```
use Throwable;

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

Dynamic Blade Components

Dynamic Blade components were contributed by Taylor Otwell.

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

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

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

Event Listener Improvements

Event listener improvements were contributed by Taylor Otwell.

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

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

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

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

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

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

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

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

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

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

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

Time Testing Helpers

Time testing helpers were contributed by Taylor Otwell with inspiration from Ruby on Rails.

When testing, you may occasionally need to modify the time returned by helpers such as `now` or `Illuminate\Support\Carbon::now()`. Laravel's base feature test class now includes helpers that allow you to manipulate the current time:

```
public function testTimeCanBeManipulated()
{
    // Travel into the future...
    $this->travel(5)->milliseconds();
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // Travel into the past...
    $this->travel(-5)->hours();

    // Travel to an explicit time...
    $this->travelTo(now()->subHours(6));

    // Return back to the present time...
    $this->travelBack();
}
```

Artisan `serve` Improvements

Artisan `serve` improvements were contributed by Taylor Otwell.

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

Tailwind Pagination Views

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

Routing Namespace Updates

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

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

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

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

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

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

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

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

Laravel Sail

Introduction

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

At its heart, Sail is the `docker-compose.yml` file and the `sail` script that is stored at the root of your project. The `sail` script provides a CLI with convenient methods for interacting with the Docker containers defined by the `docker-compose.yml` file.

Laravel Sail is supported on macOS, Linux, and Windows (via WSL2).

Installation & Setup

Laravel Sail is automatically installed with all new Laravel applications so you may start using it immediately. To learn how to create a new Laravel application, please consult Laravel's installation documentation for your operating system. During installation, you will be asked to choose which Sail supported services your application will be interacting with.

Installing Sail Into Existing Applications

If you are interested in using Sail with an existing Laravel application, you may simply install Sail using the Composer package manager. Of course, these steps assume that your existing local development environment allows you to install Composer dependencies:

```
composer require laravel/sail --dev
```

After Sail has been installed, you may run the `sail:install` Artisan command. This command will publish Sail's `docker-compose.yml` file to the root of your application:

```
php artisan sail:install
```

Finally, you may start Sail. To continue learning how to use Sail, please continue reading the remainder of this documentation:

```
./vendor/bin/sail up
```

Using Devcontainers

If you would like to develop within a Devcontainer, you may provide the `--devcontainer` option to the `sail:install` command. The `--devcontainer` option will instruct the `sail:install` command to publish a default `.devcontainer/devcontainer.json` file to the root of your application:

```
php artisan sail:install --devcontainer
```

Configuring A Bash Alias

By default, Sail commands are invoked using the `vendor/bin/sail` script that is included with all new Laravel applications:

```
./vendor/bin/sail up
```

However, instead of repeatedly typing `vendor/bin/sail` to execute Sail commands, you may wish to configure a Bash alias that allows you to execute Sail's commands more easily:

```
alias sail='[ -f sail ] && bash sail || bash vendor/bin/sail'
```

Once the Bash alias has been configured, you may execute Sail commands by simply typing `sail`. The remainder of this documentation's examples will assume that you have configured this alias:

```
sail up
```

Starting & Stopping Sail

Laravel Sail's `docker-compose.yml` file defines a variety of Docker containers that work together to help you build Laravel applications. Each of these containers is an entry within the `services` configuration of your `docker-compose.yml` file. The `laravel.test` container is the primary application container that will be serving your application.

Before starting Sail, you should ensure that no other web servers or databases are running on your local computer. To start all of the Docker containers defined in your application's `docker-compose.yml` file, you should execute the `up` command:

```
sail up
```

To start all of the Docker containers in the background, you may start Sail in "detached" mode:

```
sail up -d
```

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

To stop all of the containers, you may simply press Control + C to stop the container's execution. Or, if the containers are running in the background, you may use the `stop` command:

```
sail stop
```

Executing Commands

When using Laravel Sail, your application is executing within a Docker container and is isolated from your local computer. However, Sail provides a convenient way to run various commands against your application such as arbitrary PHP commands, Artisan commands, Composer commands, and Node / NPM commands.

When reading the Laravel documentation, you will often see references to Composer, Artisan, and Node / NPM commands that do not reference Sail. Those examples assume that these tools are installed on your local computer. If you are using Sail for your local Laravel development environment, you should execute those commands using Sail:

```
# Running Artisan commands locally...
php artisan queue:work

# Running Artisan commands within Laravel Sail...
sail artisan queue:work
```

Executing PHP Commands

PHP commands may be executed using the `php` command. Of course, these commands will execute using the PHP version that is configured for your application. To learn more about the PHP versions available to Laravel Sail, consult the PHP version documentation:

```
sail php --version

sail php script.php
```

Executing Composer Commands

Composer commands may be executed using the `composer` command. Laravel Sail's application container includes a Composer 2.x installation:

```
sail composer require laravel/sanctum
```

Installing Composer Dependencies For Existing Applications

If you are developing an application with a team, you may not be the one that initially creates the Laravel application. Therefore, none of the application's Composer dependencies, including Sail, will be installed after you clone the application's repository to your local computer.

You may install the application's dependencies by navigating to the application's directory and executing the following command. This command uses a small Docker container containing PHP and Composer to install the application's dependencies:

```
docker run --rm \
    -u "$(id -u):$(id -g)" \
    -v $(pwd):/var/www/html \
    -w /var/www/html \
    laravelsail/php80-composer:latest \
    composer install --ignore-platform-reqs
```

Executing Artisan Commands

Laravel Artisan commands may be executed using the `artisan` command:

```
sail artisan queue:work
```

Executing Node / NPM Commands

Node commands may be executed using the `node` command while NPM commands may be executed using the `npm` command:

```
sail node --version

sail npm run prod
```

Interacting With Databases

MySQL

As you may have noticed, your application's `docker-compose.yml` file contains an entry for a MySQL container. This container uses a Docker volume so that the data stored in your database is persisted even when stopping and restarting your containers. In addition, when the MySQL container is starting, it will ensure a database exists whose name matches the value of your `DB_DATABASE` environment variable.

Once you have started your containers, you may connect to the MySQL instance within your application by setting your `DB_HOST` environment variable within your application's `.env` file to `mysql`.

To connect to your application's MySQL database from your local machine, you may use a graphical database management application such as TablePlus. By default, the MySQL database is accessible at `localhost` port 3306.

Redis

Your application's `docker-compose.yml` file also contains an entry for a Redis container. This container uses a Docker volume so that the data stored in your Redis data is persisted even when stopping and restarting your containers. Once you have started your containers, you may connect to the Redis instance within your application by setting your `REDIS_HOST` environment variable within your application's `.env` file to `redis`.

To connect to your application's Redis database from your local machine, you may use a graphical database management application such as TablePlus. By default, the Redis database is accessible at `localhost` port 6379.

MeiliSearch

If you chose to install the MeiliSearch service when installing Sail, your application's `docker-compose.yml` file will contain an entry for this powerful search-engine that is compatible with Laravel Scout. Once you have started your containers, you may connect to the MeiliSearch instance within your application by setting your `MEILISEARCH_HOST` environment variable to `http://meilisearch:7700`.

From your local machine, you may access MeiliSearch's web based administration panel by navigating to `http://localhost:7700` in your web browser.

File Storage

If you plan to use Amazon S3 to store files while running your application in its production environment, you may wish to install the MinIO service when installing Sail. MinIO provides an S3 compatible API that you may use to develop locally using Laravel's `s3` file storage driver without creating "test" storage buckets in your production S3 environment. If you choose to install MinIO while installing Sail, a MinIO configuration section will be added to your application's `docker-compose.yml` file.

By default, your application's `filesystems` configuration file already contains a disk configuration for the `s3` disk. In addition to using this disk to interact with Amazon S3, you may use it to interact with any S3 compatible file storage service such as MinIO by simply modifying the associated environment variables that control its configuration. For example, when using MinIO, your filesystem environment variable configuration should be defined as follows:

```
FILESYSTEM_DRIVER=s3
AWS_ACCESS_KEY_ID=sail
AWS_SECRET_ACCESS_KEY=password
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=local
AWS_ENDPOINT=http://minio:9000
AWS_USE_PATH_STYLE_ENDPOINT=true
```

Running Tests

Laravel provides amazing testing support out of the box, and you may use Sail's `test` command to run your applications feature and unit tests. Any CLI options that are accepted by PHPUnit may also be passed to the `test` command:

```
sail test

sail test --group orders
```

The Sail `test` command is equivalent to running the `test` Artisan command:

```
sail artisan test
```

Laravel Dusk

Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. Thanks to Sail, you may run these tests without ever installing Selenium or other tools on your local computer. To get started, uncomment the Selenium service in your application's `docker-compose.yml` file:

```
selenium:
    image: 'selenium/standalone-chrome'
    volumes:
        - '/dev/shm:/dev/shm'
    networks:
        - sail
```

Next, ensure that the `laravel.test` service in your application's `docker-compose.yml` file has a `depends_on` entry for `selenium`:

```
depends_on:
    - mysql
    - redis
    - selenium
```

Finally, you may run your Dusk test suite by starting Sail and running the `dusk` command:

```
sail dusk
```

Selenium On Apple Silicon

If your local machine contains an Apple Silicon chip, your `selenium` service must use the `seleniarm/standalone-chromium` image:

```
selenium:
    image: 'seleniarm/standalone-chromium'
    volumes:
        - '/dev/shm:/dev/shm'
    networks:
        - sail
```

Previewing Emails

Laravel Sail's default `docker-compose.yml` file contains a service entry for MailHog. MailHog intercepts emails sent by your application during local development and provides a convenient web interface so that you can preview your email messages in your browser. When using Sail, MailHog's default host is `mailhog` and is available via port 1025:

```
MAIL_HOST=mailhog
MAIL_PORT=1025
MAIL_ENCRYPTION=null
```

When Sail is running, you may access the MailHog web interface at: http://localhost:8025

Container CLI

Sometimes you may wish to start a Bash session within your application's container. You may use the `shell` command to connect to your application's container, allowing you to inspect its files and installed services as well execute arbitrary shell commands within the container:

```
sail shell

sail root-shell
```

To start a new Laravel Tinker session, you may execute the `tinker` command:

```
sail tinker
```

PHP Versions

Sail currently supports serving your application via PHP 8.1, PHP 8.0, or PHP 7.4. The default PHP version used by Sail is currently PHP 8.1. To change the PHP version that is used to serve your application, you should update the `build` definition of the `laravel.test` container in your application's `docker-compose.yml` file:

```
# PHP 8.1
context: ./vendor/laravel/sail/runtimes/8.1

# PHP 8.0
context: ./vendor/laravel/sail/runtimes/8.0

# PHP 7.4
context: ./vendor/laravel/sail/runtimes/7.4
```

In addition, you may wish to update your `image` name to reflect the version of PHP being used by your application. This option is also defined in your application's `docker-compose.yml` file:

```
image: sail-8.1/app
```

After updating your application's `docker-compose.yml` file, you should rebuild your container images:

```
sail build --no-cache

sail up
```

Sharing Your Site

Sometimes you may need to share your site publicly in order to preview your site for a colleague or to test webhook integrations with your application. To share your site, you may use the `share` command. After executing this command, you will be issued a random `laravel-sail.site` URL that you may use to access your application:

```
sail share
```

When sharing your site via the `share` command, you should configure your application's trusted proxies within the `TrustProxies` middleware. Otherwise, URL generation helpers such as `url` and `route` will be unable to determine the correct HTTP host that should be used during URL generation:

```
/**
 * The trusted proxies for this application.
 *
 * @var array|string|null
 */
protected $proxies = '*';
```

If you would like to choose the subdomain for your shared site, you may provide the `subdomain` option when executing the `share` command:

```
sail share --subdomain=my-sail-site
```

{tip} The `share` command is powered by Expose, an open source tunneling service by BeyondCode.

Debugging With Xdebug

Laravel Sail's Docker configuration includes support for Xdebug, a popular and powerful debugger for PHP. In order to enable Xdebug, you will need to add a few variables to your application's `.env` file to configure Xdebug. To enable Xdebug you must set the appropriate mode(s) before starting Sail:

```
SAIL_XDEBUG_MODE=develop,debug
```

Linux Host IP Configuration

Internally, the `XDEBUG_CONFIG` environment variable is defined as `client_host=host.docker.internal` so that Xdebug will be properly configured for Mac and Windows (WSL2). If your local machine is running Linux, you will need to manually define this environment variable.

First, you should determine the correct host IP address to add to the environment variable by running the following command. Typically, the `<container-name>` should be the name of the container that serves your application and often ends with `_laravel.test_1`:

```
docker inspect -f {{range.NetworkSettings.Networks}}{{.Gateway}}{{end}}
<container-name>
```

Once you have obtained the correct host IP address, you should define the `SAIL_XDEBUG_CONFIG` variable within your application's `.env` file:

```
SAIL_XDEBUG_CONFIG="client_host=<host-ip-address>"
```

Xdebug CLI Usage

A `sail debug` command may be used to start a debugging session when running an Artisan command:

```
# Run an Artisan command without Xdebug...
sail artisan migrate

# Run an Artisan command with Xdebug...
sail debug migrate
```

Xdebug Browser Usage

To debug your application while interacting with the application via a web browser, follow the instructions provided by Xdebug for initiating an Xdebug session from the web browser.

If you're using PhpStorm, please review JetBrain's documentation regarding zero-configuration debugging.

{note} Laravel Sail relies on `artisan serve` to serve your application. The `artisan serve` command only accepts the `XDEBUG_CONFIG` and `XDEBUG_MODE` variables as of Laravel version 8.53.0. Older versions of Laravel (8.52.0 and below) do not support these variables and will not accept debug connections.

Customization

Since Sail is just Docker, you are free to customize nearly everything about it. To publish Sail's own Dockerfiles, you may execute the `sail:publish` command:

```
sail artisan sail:publish
```

After running this command, the Dockerfiles and other configuration files used by Laravel Sail will be placed within a `docker` directory in your application's root directory. After customizing your Sail installation, you may wish to change the image name for the `laravel.test` service in your application's `docker-compose.yml` file. After doing so, rebuild your application's containers using the `build` command. Assigning a unique name to the `laravel.test` service image is particularly important if you are using Sail to develop multiple Laravel applications on a single machine:

```
sail build --no-cache
```

Laravel Sanctum

- Testing

Introduction

Laravel Sanctum provides a featherweight authentication system for SPAs (single page applications), mobile applications, and simple, token based APIs. Sanctum allows each user of your application to generate multiple API tokens for their account. These tokens may be granted abilities / scopes which specify which actions the tokens are allowed to perform.

How It Works

Laravel Sanctum exists to solve two separate problems. Let's discuss each before digging deeper into the library.

API Tokens

First, Sanctum is a simple package you may use to issue API tokens to your users without the complication of OAuth. This feature is inspired by GitHub and other applications which issue "personal access tokens". For example, imagine the "account settings" of your application has a screen where a user may generate an API token for their account. You may use Sanctum to generate and manage those tokens. These tokens typically have a very long expiration time (years), but may be manually revoked by the user at anytime.

Laravel Sanctum offers this feature by storing user API tokens in a single database table and authenticating incoming HTTP requests via the `Authorization` header which should contain a valid API token.

SPA Authentication

Second, Sanctum exists to offer a simple way to authenticate single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository, such as a SPA created using Vue CLI or a Next.js application.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. Typically, Sanctum utilizes Laravel's `web` authentication guard to accomplish this. This provides the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.

Sanctum will only attempt to authenticate using cookies when the incoming request originates from your own SPA frontend. When Sanctum examines an incoming HTTP request,

it will first check for an authentication cookie and, if none is present, Sanctum will then examine the `Authorization` header for a valid API token.

{tip} It is perfectly fine to use Sanctum only for API token authentication or only for SPA authentication. Just because you use Sanctum does not mean you are required to use both features it offers.

Installation

{tip} The most recent versions of Laravel already include Laravel Sanctum. However, if your application's `composer.json` file does not include `laravel/sanctum`, you may follow the installation instructions below.

You may install Laravel Sanctum via the Composer package manager:

```
composer require laravel/sanctum
```

Next, you should publish the Sanctum configuration and migration files using the `vendor:publish` Artisan command. The `sanctum` configuration file will be placed in your application's `config` directory:

```
php artisan vendor:publish --
provider="Laravel\Sanctum\SanctumServiceProvider"
```

Finally, you should run your database migrations. Sanctum will create one database table in which to store API tokens:

```
php artisan migrate
```

Next, if you plan to utilize Sanctum to authenticate an SPA, you should add Sanctum's middleware to your `api` middleware group within your application's `app/Http/Kernel.php` file:

```
'api' => [
\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::clas
s,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

Migration Customization

If you are not going to use Sanctum's default migrations, you should call the `Sanctum::ignoreMigrations` method in the `register` method of your `App\Providers\AppServiceProvider` class. You may export the default migrations by executing the following command: `php artisan vendor:publish --tag=sanctum-migrations`

Configuration

Overriding Default Models

Although not typically required, you are free to extend the `PersonalAccessToken` model used internally by Sanctum:

```
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;

class PersonalAccessToken extends SanctumPersonalAccessToken
{
    // ...
}
```

Then, you may instruct Sanctum to use your custom model via the `usePersonalAccessTokenModel` method provided by Sanctum. Typically, you should call this method in the `boot` method of one of your application's service providers:

```
use App\Models\Sanctum\PersonalAccessToken;
use Laravel\Sanctum\Sanctum;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Sanctum::usePersonalAccessTokenModel(PersonalAccessToken::class);
}
```

API Token Authentication

{tip} You should not use API tokens to authenticate your own first-party SPA. Instead, use Sanctum's built-in SPA authentication features.

Issuing API Tokens

Sanctum allows you to issue API tokens / personal access tokens that may be used to authenticate API requests to your application. When making requests using API tokens, the token should be included in the `Authorization` header as a `Bearer` token.

To begin issuing tokens for users, your User model should use the `Laravel\Sanctum\HasApiTokens` trait:

```
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

To issue a token, you may use the `createToken` method. The `createToken` method returns a `Laravel\Sanctum\NewAccessToken` instance. API tokens are hashed using SHA-256 hashing before being stored in your database, but you may access the plain-text value of the token using the `plainTextToken` property of the `NewAccessToken` instance. You should display this value to the user immediately after the token has been created:

```
use Illuminate\Http\Request;

Route::post('/tokens/create', function (Request $request) {
    $token = $request->user()->createToken($request->token_name);

    return ['token' => $token->plainTextToken];
});
```

You may access all of the user's tokens using the `tokens` Eloquent relationship provided by the `HasApiTokens` trait:

```
foreach ($user->tokens as $token) {
    //
}
```

Token Abilities

Sanctum allows you to assign "abilities" to tokens. Abilities serve a similar purpose as OAuth's "scopes". You may pass an array of string abilities as the second argument to the `createToken` method:

```
return $user->createToken('token-name',
['server:update'])->plainTextToken;
```

When handling an incoming request authenticated by Sanctum, you may determine if the token has a given ability using the `tokenCan` method:

```
if ($user->tokenCan('server:update')) {
    //
}
```

Token Ability Middleware

Sanctum also includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given ability. To get started, add the following middleware to the `$routeMiddleware` property of your application's `app/Http/Kernel.php` file:

```
'abilities' => \Laravel\Sanctum\Http\Middleware\CheckAbilities::class,
'ability' => \Laravel\Sanctum\Http\Middleware\CheckForAnyAbility::class,
```

The `abilities` middleware may be assigned to a route to verify that the incoming request's token has all of the listed abilities:

```
Route::get('/orders', function () {
    // Token has both "check-status" and "place-orders" abilities...
})->middleware(['auth:sanctum', 'abilities:check-status,place-orders']);
```

The `ability` middleware may be assigned to a route to verify that the incoming request's token has *at least one* of the listed abilities:

```
Route::get('/orders', function () {
    // Token has the "check-status" or "place-orders" ability...
})->middleware(['auth:sanctum', 'ability:check-status,place-orders']);
```

First-Party UI Initiated Requests

For convenience, the `tokenCan` method will always return `true` if the incoming authenticated request was from your first-party SPA and you are using Sanctum's built-in SPA authentication.

However, this does not necessarily mean that your application has to allow the user to perform the action. Typically, your application's authorization policies will determine if the token has been granted the permission to perform the abilities as well as check that the user instance itself should be allowed to perform the action.

For example, if we imagine an application that manages servers, this might mean checking that token is authorized to update servers **and** that the server belongs to the user:

```
return $request->user()->id === $server->user_id &&
       $request->user()->tokenCan('server:update')
```

At first, allowing the `tokenCan` method to be called and always return `true` for first-party UI initiated requests may seem strange; however, it is convenient to be able to always assume an API token is available and can be inspected via the `tokenCan` method. By taking this approach, you may always call the `tokenCan` method within your application's authorizations policies without worrying about whether the request was triggered from your application's UI or was initiated by one of your API's third-party consumers.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your protected routes within your `routes/web.php` and `routes/api.php` route files. This guard will ensure that incoming requests are authenticated as either stateful, cookie authenticated requests or contain a valid API token header if the request is from a third party.

You may be wondering why we suggest that you authenticate the routes within your application's `routes/web.php` file using the `sanctum` guard. Remember, Sanctum will first attempt to authenticate incoming requests using Laravel's typical session authentication cookie. If that cookie is not present then Sanctum will attempt to authenticate the request using a token in the request's `Authorization` header. In addition, authenticating all requests using Sanctum ensures that we may always call the `tokenCan` method on the currently authenticated user instance:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request
$request) {
    return $request->user();
});
```

Revoking Tokens

You may "revoke" tokens by deleting them from your database using the `tokens` relationship that is provided by the `Laravel\Sanctum\HasApiTokens` trait:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke the token that was used to authenticate the current request...
$request->user()->currentAccessToken()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

SPA Authentication

Sanctum also exists to provide a simple method of authenticating single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. This approach to authentication provides the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.

{note} In order to authenticate, your SPA and API must share the same top-level domain. However, they may be placed on different subdomains. Additionally, you should ensure that you send the `Accept: application/json` header with your request.

Configuration

Configuring Your First-Party Domains

First, you should configure which domains your SPA will be making requests from. You may configure these domains using the `stateful` configuration option in your `sanctum` configuration file. This configuration setting determines which domains will maintain "stateful" authentication using Laravel session cookies when making requests to your API.

{note} If you are accessing your application via a URL that includes a port (`127.0.0.1:8000`), you should ensure that you include the port number with the domain.

Sanctum Middleware

Next, you should add Sanctum's middleware to your `api` middleware group within your `app/Http/Kernel.php` file. This middleware is responsible for ensuring that incoming requests from your SPA can authenticate using Laravel's session cookies, while still allowing requests from third parties or mobile applications to authenticate using API tokens:

```
'api' => [
\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::clas
s,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

CORS & Cookies

If you are having trouble authenticating with your application from a SPA that executes on a separate subdomain, you have likely misconfigured your CORS (Cross-Origin Resource Sharing) or session cookie settings.

You should ensure that your application's CORS configuration is returning the `Access-Control-Allow-Credentials` header with a value of `True`. This may be accomplished by setting the `supports_credentials` option within your application's `config/cors.php` configuration file to `true`.

In addition, you should enable the `withCredentials` option on your application's global `axios` instance. Typically, this should be performed in your `resources/js/bootstrap.js` file. If you are not using Axios to make HTTP requests from your frontend, you should perform the equivalent configuration on your own HTTP client:

```
axios.defaults.withCredentials = true;
```

Finally, you should ensure your application's session cookie domain configuration supports any subdomain of your root domain. You may accomplish this by prefixing the domain with a leading `.` within your application's `config/session.php` configuration file:

```
'domain' => '.domain.com',
```

Authenticating

CSRF Protection

To authenticate your SPA, your SPA's "login" page should first make a request to the `/sanctum/csrf-cookie` endpoint to initialize CSRF protection for the application:

```
axios.get('/sanctum/csrf-cookie').then(response => {
    // Login...
});
```

During this request, Laravel will set an `XSRF-TOKEN` cookie containing the current CSRF token. This token should then be passed in an `X-XSRF-TOKEN` header on subsequent requests, which some HTTP client libraries like Axios and the Angular HttpClient will do automatically for you. If your JavaScript HTTP library does not set the value for you, you will need to manually set the `X-XSRF-TOKEN` header to match the value of the `XSRF-TOKEN` cookie that is set by this route.

Logging In

Once CSRF protection has been initialized, you should make a `POST` request to your Laravel application's `/login` route. This `/login` route may be implemented manually or using a headless authentication package like Laravel Fortify.

If the login request is successful, you will be authenticated and subsequent requests to your application's routes will automatically be authenticated via the session cookie that the Laravel application issued to your client. In addition, since your application already made a request to the `/sanctum/csrf-cookie` route, subsequent requests should automatically receive CSRF protection as long as your JavaScript HTTP client sends the value of the `XSRF-TOKEN` cookie in the `X-XSRF-TOKEN` header.

Of course, if your user's session expires due to lack of activity, subsequent requests to the Laravel application may receive 401 or 419 HTTP error response. In this case, you should redirect the user to your SPA's login page.

{note} You are free to write your own `/login` endpoint; however, you should ensure that it authenticates the user using the standard, session based authentication services that Laravel provides. Typically, this means using the `web` authentication guard.

Protecting Routes

To protect routes so that all incoming requests must be authenticated, you should attach the `sanctum` authentication guard to your API routes within your `routes/api.php` file. This guard will ensure that incoming requests are authenticated as either a stateful authenticated requests from your SPA or contain a valid API token header if the request is from a third party:

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request
$request) {
    return $request->user();
});
```

Authorizing Private Broadcast Channels

If your SPA needs to authenticate with private / presence broadcast channels, you should place the `Broadcast::routes` method call within your `routes/api.php` file:

```
Broadcast::routes(['middleware' => ['auth:sanctum']]);
```

Next, in order for Pusher's authorization requests to succeed, you will need to provide a custom Pusher `authorizer` when initializing Laravel Echo. This allows your application to configure Pusher to use the `axios` instance that is properly configured for cross-domain requests:

```
window.Echo = new Echo({
    broadcaster: "pusher",
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    encrypted: true,
    key: process.env.MIX_PUSHER_APP_KEY,
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(false, response.data);
                })
                .catch(error => {
                    callback(true, error);
                });
            }
        };
    },
})
```

Mobile Application Authentication

You may also use Sanctum tokens to authenticate your mobile application's requests to your API. The process for authenticating mobile application requests is similar to authenticating third-party API requests; however, there are small differences in how you will issue the API tokens.

Issuing API Tokens

To get started, create a route that accepts the user's email / username, password, and device name, then exchanges those credentials for a new Sanctum token. The "device name" given to this endpoint is for informational purposes and may be any value you wish. In general, the device name value should be a name the user would recognize, such as "Nuno's iPhone 12".

Typically, you will make a request to the token endpoint from your mobile application's "login" screen. The endpoint will return the plain-text API token which may then be stored on the mobile device and used to make additional API requests:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

Route::post('/sanctum/token', function (Request $request) {
    $request->validate([
        'email' => 'required|email',
        'password' => 'required',
        'device_name' => 'required',
    ]);

    $user = User::where('email', $request->email)->first();

    if (! $user || ! Hash::check($request->password, $user->password)) {
        throw ValidationException::withMessages([
            'email' => ['The provided credentials are incorrect.'],
        ]);
    }

    return $user->createToken($request->device_name)->plainTextToken;
});
```

When the mobile application uses the token to make an API request to your application, it should pass the token in the `Authorization` header as a `Bearer` token.

{tip} When issuing tokens for a mobile application, you are also free to specify token abilities.

Protecting Routes

As previously documented, you may protect routes so that all incoming requests must be authenticated by attaching the `sanctum` authentication guard to the routes:

```
Route::middleware('auth:sanctum')->get('/user', function (Request
$request) {
    return $request->user();
});
```

Revoking Tokens

To allow users to revoke API tokens issued to mobile devices, you may list them by name, along with a "Revoke" button, within an "account settings" portion of your web application's UI. When the user clicks the "Revoke" button, you can delete the token from the database. Remember, you can access a user's API tokens via the `tokens` relationship provided by the `Laravel\Sanctum\HasApiTokens` trait:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

Testing

While testing, the `Sanctum::actingAs` method may be used to authenticate a user and specify which abilities should be granted to their token:

```
use App\Models\User;
use Laravel\Sanctum\Sanctum;

public function test_task_list_can_be_retrieved()
{
    Sanctum::actingAs(
        User::factory()->create(),
        ['view-tasks']
    );

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

    $response->assertOk();
}
```

If you would like to grant all abilities to the token, you should include `*` in the ability list provided to the `actingAs` method:

```
Sanctum::actingAs(
    User::factory()->create(),
    ['*']
);
```

Task Scheduling

Introduction

In the past, you may have written a cron configuration entry for each task you needed to schedule on your server. However, this can quickly become a pain because your task schedule is no longer in source control and you must SSH into your server to view your existing cron entries or add additional entries.

Laravel's command scheduler offers a fresh approach to managing scheduled tasks on your server. The scheduler allows you to fluently and expressively define your command schedule within your Laravel application itself. When using the scheduler, only a single cron entry is needed on your server. Your task schedule is defined in the `app/Console/Kernel.php` file's `schedule` method. To help you get started, a simple example is defined within the method.

Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of your application's `App\Console\Kernel` class. To get started, let's take a look at an example. In this example, we will schedule a closure to be called every day at midnight. Within the closure we will execute a database query to clear a table:

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

In addition to scheduling using closures, you may also schedule invokable objects. Invokable objects are simple PHP classes that contain an `__invoke` method:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

If you would like to view an overview of your scheduled tasks and the next time they are scheduled to run, you may use the `schedule:list` Artisan command:

```
php artisan schedule:list
```

Scheduling Artisan Commands

In addition to scheduling closures, you may also schedule Artisan commands and system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class.

When scheduling Artisan commands using the command's class name, you may pass an array of additional command-line arguments that should be provided to the command when it is invoked:

```
use App\Console\Commands\SendEmailsCommand;

$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(SendEmailsCommand::class, ['Taylor', '--
force'])->daily();
```

Scheduling Queued Jobs

The `job` method may be used to schedule a queued job. This method provides a convenient way to schedule queued jobs without using the `call` method to define closures to queue the job:

```
use App\Jobs\Heartbeat;

$schedule->job(new Heartbeat)->everyFiveMinutes();
```

Optional second and third arguments may be provided to the `job` method which specifies the queue name and queue connection that should be used to queue the job:

```
use App\Jobs\Heartbeat;

// Dispatch the job to the "heartbeats" queue on the "sqs" connection...
$schedule->job(new Heartbeat, 'heartbeats', 'sqs')->everyFiveMinutes();
```

Scheduling Shell Commands

The `exec` method may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

We've already seen a few examples of how you may configure a task to run at specified intervals. However, there are many more task schedule frequencies that you may assign to a task:

Method	Description
`->cron('* * * * *');`	Run the task on a custom cron schedule
`->everyMinute();`	Run the task every minute
`->everyTwoMinutes();`	Run the task every two minutes
`->everyThreeMinutes();`	Run the task every three minutes
`->everyFourMinutes();`	Run the task every four minutes
`->everyFiveMinutes();`	Run the task every five minutes
`->everyTenMinutes();`	Run the task every ten minutes
`->everyFifteenMinutes();`	Run the task every fifteen minutes
`->everyThirtyMinutes();`	Run the task every thirty minutes
`->hourly();`	Run the task every hour

Method	Description
`->hourlyAt(17);`	Run the task every hour at 17 minutes past the hour
`->everyTwoHours();`	Run the task every two hours
`->everyThreeHours();`	Run the task every three hours
`->everyFourHours();`	Run the task every four hours
`->everySixHours();`	Run the task every six hours
`->daily();`	Run the task every day at midnight
`->dailyAt('13:00');`	Run the task every day at 13:00
`->twiceDaily(1, 13);`	Run the task daily at 1:00 & 13:00
`->weekly();`	Run the task every Sunday at 00:00
`->weeklyOn(1, '8:00');`	Run the task every week on Monday at 8:00
`->monthly();`	Run the task on the first day of every month at 00:00
`->monthlyOn(4, '15:00');`	Run the task every month on the 4th at 15:00
`->twiceMonthly(1, 16, '13:00');`	Run the task monthly on the 1st and 16th at 13:00
`->lastDayOfMonth('15:00');`	Run the task on the last day of the month at 15:00
`->quarterly();`	Run the task on the first day of every quarter at 00:00
`->yearly();`	Run the task on the first day of every year at 00:00
`->yearlyOn(6, 1, '17:00');`	Run the task every year on June 1st at 17:00
`->timezone('America/New_York');`	Set the timezone for the task

These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, you may schedule a command to run weekly on Monday:

```
// Run once per week on Monday at 1 PM...
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')
          ->weekdays()
          ->hourly()
          ->timezone('America/Chicago')
          ->between('8:00', '17:00');
```

A list of additional schedule constraints may be found below:

Method	Description
`->weekdays();`	Limit the task to weekdays
`->weekends();`	Limit the task to weekends
`->sundays();`	Limit the task to Sunday
`->mondays();`	Limit the task to Monday
`->tuesdays();`	Limit the task to Tuesday
`->wednesdays();`	Limit the task to Wednesday
`->thursdays();`	Limit the task to Thursday
`->fridays();`	Limit the task to Friday
`->saturdays();`	Limit the task to Saturday
`->days(array	mixed);`
`->between($startTime, $endTime);`	Limit the task to run between start and end times
`->unlessBetween($startTime, $endTime);`	Limit the task to not run between start and end times
`->when(Closure);`	Limit the task based on a truth test
`->environments($env);`	Limit the task to specific environments

Day Constraints

The `days` method may be used to limit the execution of a task to specific days of the week. For example, you may schedule a command to run hourly on Sundays and Wednesdays:

```
$schedule->command('emails:send')
                ->hourly()
                ->days([0, 3]);
```

Alternatively, you may use the constants available on the `Illuminate\Console\Scheduling\Schedule` class when defining the days on which a task should run:

```
use Illuminate\Console\Scheduling\Schedule;

$schedule->command('emails:send')
                ->hourly()
                ->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

Between Time Constraints

The `between` method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('emails:send')
                    ->hourly()
                    ->between('7:00', '22:00');
```

Similarly, the `unlessBetween` method can be used to exclude the execution of a task for a period of time:

```
$schedule->command('emails:send')
                    ->hourly()
                    ->unlessBetween('23:00', '4:00');
```

Truth Test Constraints

The `when` method may be used to limit the execution of a task based on the result of a given

truth test. In other words, if the given closure returns `true`, the task will execute as long as no other constraining conditions prevent the task from running:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

The `skip` method may be seen as the inverse of `when`. If the `skip` method returns `true`, the scheduled task will not be executed:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return `true`.

Environment Constraints

The `environments` method may be used to execute tasks only on the given environments (as defined by the `APP_ENV` environment variable):

```
$schedule->command('emails:send')
            ->daily()
            ->environments(['staging', 'production']);
```

Timezones

Using the `timezone` method, you may specify that a scheduled task's time should be interpreted within a given timezone:

```
$schedule->command('report:generate')
         ->timezone('America/New_York')
         ->at('2:00')
```

If you are repeatedly assigning the same timezone to all of your scheduled tasks, you may wish to define a `scheduleTimezone` method in your `App\Console\Kernel` class. This method should return the default timezone that should be assigned to all scheduled tasks:

```
/**
 * Get the timezone that should be used by default for scheduled events.
 *
 * @return \DateTimeZone|string|null
 */
protected function scheduleTimezone()
{
    return 'America/Chicago';
}
```

{note} Remember that some timezones utilize daylight savings time. When daylight saving time changes occur, your scheduled task may run twice or even not run at all. For this reason, we recommend avoiding timezone scheduling when possible.

Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` Artisan command will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

If needed, you may specify how many minutes must pass before the "without overlapping" lock expires. By default, the lock will expire after 24 hours:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

Running Tasks On One Server

{note} To utilize this feature, your application must be using the `database`, `memcached`, `dynamodb`, or `redis` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

If your application's scheduler is running on multiple servers, you may limit a scheduled job to only execute on a single server. For instance, assume you have a scheduled task that generates a new report every Friday night. If the task scheduler is running on three worker servers, the scheduled task will run on all three servers and generate the report three times. Not good!

To indicate that the task should run on only one server, use the `onOneServer` method when defining the scheduled task. The first server to obtain the task will secure an atomic lock on the job to prevent other servers from running the same task at the same time:

```
$schedule->command('report:generate')
                 ->fridays()
                 ->at('17:00')
                 ->onOneServer();
```

Background Tasks

By default, multiple tasks scheduled at the same time will execute sequentially based on the order they are defined in your `schedule` method. If you have long-running tasks, this may cause subsequent tasks to start much later than anticipated. If you would like to run tasks in the background so that they may all run simultaneously, you may use the `runInBackground` method:

```
$schedule->command('analytics:report')
          ->daily()
          ->runInBackground();
```

{note} The `runInBackground` method may only be used when scheduling tasks via the `command` and `exec` methods.

Maintenance Mode

Your application's scheduled tasks will not run when the application is in maintenance mode, since we don't want your tasks to interfere with any unfinished maintenance you may be performing on your server. However, if you would like to force a task to run even in maintenance mode, you may call the `evenInMaintenanceMode` method when defining the task:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

Running The Scheduler

Now that we have learned how to define scheduled tasks, let's discuss how to actually run them on our server. The `schedule:run` Artisan command will evaluate all of your scheduled tasks and determine if they need to run based on the server's current time.

So, when using Laravel's scheduler, we only need to add a single cron configuration entry to our server that runs the `schedule:run` command every minute. If you do not know how to add cron entries to your server, consider using a service such as Laravel Forge which can manage the cron entries for you:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >>
/dev/null 2>&1
```

Running The Scheduler Locally

Typically, you would not add a scheduler cron entry to your local development machine. Instead, you may use the `schedule:work` Artisan command. This command will run in the foreground and invoke the scheduler every minute until you terminate the command:

```
php artisan schedule:work
```

Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
         ->daily()
         ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
         ->daily()
         ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may email the output to an email address of your choice. Before emailing the output of a task, you should configure Laravel's email services:

```
$schedule->command('report:generate')
         ->daily()
         ->sendOutputTo($filePath)
         ->emailOutputTo('taylor@example.com');
```

If you only want to email the output if the scheduled Artisan or system command terminates with a non-zero exit code, use the `emailOutputOnFailure` method:

```
$schedule->command('report:generate')
         ->daily()
         ->emailOutputOnFailure('taylor@example.com');
```

{note} The `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo`, and `appendOutputTo` methods are exclusive to the `command` and `exec` methods.

Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is executed:

```
$schedule->command('emails:send')
         ->daily()
         ->before(function () {
             // The task is about to execute...
         })
         ->after(function () {
             // The task has executed...
         });
```

The `onSuccess` and `onFailure` methods allow you to specify code to be executed if the scheduled task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
         ->daily()
         ->onSuccess(function () {
             // The task succeeded...
         })
         ->onFailure(function () {
             // The task failed...
         });
```

If output is available from your command, you may access it in your `after`, `onSuccess` or `onFailure` hooks by type-hinting an `Illuminate\Support\Stringable` instance as the `$output` argument of your hook's closure definition:

```
use Illuminate\Support\Stringable;

$schedule->command('emails:send')
         ->daily()
         ->onSuccess(function (Stringable $output) {
             // The task succeeded...
         })
         ->onFailure(function (Stringable $output) {
             // The task failed...
         });
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is executed. This method is useful for notifying an external service, such as Envoyer, that your scheduled task is beginning or has finished execution:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBefore($url)
         ->thenPing($url);
```

The `pingBeforeIf` and `thenPingIf` methods may be used to ping a given URL only if a given condition is `true`:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBeforeIf($condition, $url)
         ->thenPingIf($condition, $url);
```

The `pingOnSuccess` and `pingOnFailure` methods may be used to ping a given URL only if the task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
         ->daily()
         ->pingOnSuccess($successUrl)
         ->pingOnFailure($failureUrl);
```

All of the ping methods require the Guzzle HTTP library. Guzzle is typically installed in all new Laravel projects by default, but, you may manually install Guzzle into your project using the Composer package manager if it has been accidentally removed:

```
composer require guzzlehttp/guzzle
```

Events

If needed, you may listen to events dispatched by the scheduler. Typically, event listener mappings will be defined within your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Console\Events\ScheduledTaskStarting' => [
        'App\Listeners\LogScheduledTaskStarting',
    ],

    'Illuminate\Console\Events\ScheduledTaskFinished' => [
        'App\Listeners\LogScheduledTaskFinished',
    ],

    'Illuminate\Console\Events\ScheduledBackgroundTaskFinished' => [
        'App\Listeners\LogScheduledBackgroundTaskFinished',
    ],

    'Illuminate\Console\Events\ScheduledTaskSkipped' => [
        'App\Listeners\LogScheduledTaskSkipped',
    ],

    'Illuminate\Console\Events\ScheduledTaskFailed' => [
        'App\Listeners\LogScheduledTaskFailed',
    ],
];
```

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 Algolia and MeiliSearch drivers. In addition, Scout includes a "collection" driver that is designed for local development usage and does not require any external dependencies or third-party services. Furthermore, writing custom drivers is simple and you are free to extend Scout with your own search implementations.

Installation

First, install Scout via the Composer package manager:

```
composer require laravel/scout
```

After installing Scout, you should publish the Scout configuration file using the `vendor:publish` Artisan command. This command will publish the `scout.php` configuration file to your application's `config` directory:

```
php artisan vendor:publish --
provider="Laravel\Scout\ScoutServiceProvider"
```

Finally, add the `Laravel\Scout\Searchable` trait to the model you would like to make searchable. This trait will register a model observer that will automatically keep the model in sync with your search driver:

```
<?php

namespace App\Models;

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

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

Driver Prerequisites

Algolia

When using the Algolia driver, you should configure your Algolia `id` and `secret` credentials

in your `config/scout.php` configuration file. Once your credentials have been configured, you will also need to install the Algolia PHP SDK via the Composer package manager:

```
composer require algolia/algoliasearch-client-php
```

MeiliSearch

MeiliSearch is a blazingly fast and open source search engine. If you aren't sure how to install MeiliSearch on your local machine, you may use Laravel Sail, Laravel's officially supported Docker development environment.

When using the MeiliSearch driver you will need to install the MeiliSearch PHP SDK via the Composer package manager:

```
composer require meilisearch/meilisearch-php http-interop/http-factory-guzzle
```

Then, set the `SCOUT_DRIVER` environment variable as well as your MeiliSearch `host` and `key` credentials within your application's `.env` file:

```
SCOUT_DRIVER=meilisearch
MEILISEARCH_HOST=http://127.0.0.1:7700
MEILISEARCH_KEY=masterKey
```

For more information regarding MeiliSearch, please consult the MeiliSearch documentation.

In addition, you should ensure that you install a version of `meilisearch/meilisearch-php` that is compatible with your MeiliSearch binary version by reviewing MeiliSearch's documentation regarding binary compatibility.

{note} When upgrading Scout on an application that utilizes MeiliSearch, you should always review any additional breaking changes to the MeiliSearch service itself.

Queueing

While not strictly required to use Scout, you should strongly consider configuring a queue driver before using the library. Running a queue worker will allow Scout to queue all

operations that sync your model information to your search indexes, providing much better response times for your application's web interface.

Once you have configured a queue driver, set the value of the `queue` option in your `config/scout.php` configuration file to `true`:

```
'queue' => true,
```

Configuration

Configuring Model Indexes

Each Eloquent model is synced with a given search "index", which contains all of the searchable records for that model. In other words, you can think of each index like a MySQL table. By default, each model will be persisted to an index matching the model's typical "table" name. Typically, this is the plural form of the model name; however, you are free to customize the model's index by overriding the `searchableAs` method on the model:

```
<?php

namespace App\Models;

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

class Post extends Model
{
    use Searchable;

    /**
     * Get the name of the index associated with the model.
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

Configuring Searchable Data

By default, the entire `toArray` form of a given model will be persisted to its search index. If you would like to customize the data that is synchronized to the search index, you may override the `toSearchableArray` method on the model:

```php
<?php

namespace App\Models;

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

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize the data array...

        return $array;
    }
}
```

Configuring The Model ID

By default, Scout will use the primary key of the model as model's unique ID / key that is stored in the search index. If you need to customize this behavior, you may override the `getScoutKey` and the `getScoutKeyName` methods on the model:

```php
<?php

namespace App\Models;

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

class User extends Model
{
    use Searchable;

    /**
     * Get the value used to index the model.
     *
     * @return mixed
     */
    public function getScoutKey()
    {
        return $this->email;
    }

    /**
     * Get the key name used to index the model.
     *
     * @return mixed
     */
    public function getScoutKeyName()
    {
        return 'email';
    }
}
```

Identifying Users

Scout also allows you to auto identify users when using Algolia. Associating the authenticated user with search operations may be helpful when viewing your search analytics within Algolia's dashboard. You can enable user identification by defining a `SCOUT_IDENTIFY` environment variable as `true` in your application's `.env` file:

```
SCOUT_IDENTIFY=true
```

Enabling this feature this will also pass the request's IP address and your authenticated user's primary identifier to Algolia so this data is associated with any search request that is made by the user.

Local Development

While you are free to use the Algolia or MeiliSearch search engines during local development, you may find it more convenient to get started with the "collection" engine. The collection engine will use "where" clauses and collection filtering on results from your existing database to determine the applicable search results for your query. When using this engine, it is not necessary to "index" your searchable models, as they will simply be retrieved from your local database.

To use the collection engine, you may simply set the value of the `SCOUT_DRIVER` environment variable to `collection`, or specify the `collection` driver directly in your application's `scout` configuration file:

```
SCOUT_DRIVER=collection
```

Once you have specified the collection driver as your preferred driver, you may start executing search queries against your models. Search engine indexing, such as the indexing needed to seed Algolia or MeiliSearch indexes, is unnecessary when using the collection engine.

Indexing

Batch Import

If you are installing Scout into an existing project, you may already have database records you need to import into your indexes. Scout provides a `scout:import` Artisan command that you may use to import all of your existing records into your search indexes:

```
php artisan scout:import "App\Models\Post"
```

The `flush` command may be used to remove all of a model's records from your search indexes:

```
php artisan scout:flush "App\Models\Post"
```

Modifying The Import Query

If you would like to modify the query that is used to retrieve all of your models for batch importing, you may define a `makeAllSearchableUsing` method on your model. This is a great place to add any eager relationship loading that may be necessary before importing your models:

```
/**
 * Modify the query used to retrieve models when making all of the
models searchable.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $query
 * @return \Illuminate\Database\Eloquent\Builder
 */
protected function makeAllSearchableUsing($query)
{
    return $query->with('author');
}
```

Adding Records

Once you have added the `Laravel\Scout\Searchable` trait to a model, all you need to do is `save` or `create` a model instance and it will automatically be added to your search index. If you have configured Scout to use queues this operation will be performed in the background by your queue worker:

```
use App\Models\Order;

$order = new Order;

// ...

$order->save();
```

Adding Records Via Query

If you would like to add a collection of models to your search index via an Eloquent query, you may chain the `searchable` method onto the Eloquent query. The `searchable` method will chunk the results of the query and add the records to your search index. Again, if you have configured Scout to use queues, all of the chunks will be imported in the background by your queue workers:

```
use App\Models\Order;

Order::where('price', '>', 100)->searchable();
```

You may also call the `searchable` method on an Eloquent relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to add the model instances to their corresponding index:

```
$orders->searchable();
```

{tip} The `searchable` method can be considered an "upsert" operation. In other words, if the model record is already in your index, it will be updated. If it does not exist in the search index, it will be added to the index.

Updating Records

To update a searchable model, you only need to update the model instance's properties and `save` the model to your database. Scout will automatically persist the changes to your search index:

```
use App\Models\Order;

$order = Order::find(1);

// Update the order...

$order->save();
```

You may also invoke the `searchable` method on an Eloquent query instance to update a collection of models. If the models do not exist in your search index, they will be created:

```
Order::where('price', '>', 100)->searchable();
```

If you would like to update the search index records for all of the models in a relationship, you may invoke the `searchable` on the relationship instance:

```
$user->orders()->searchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `searchable` method on the collection instance to update the model instances in their corresponding index:

```
$orders->searchable();
```

Removing Records

To remove a record from your index you may simply `delete` the model from the database. This may be done even if you are using soft deleted models:

```
use App\Models\Order;

$order = Order::find(1);

$order->delete();
```

If you do not want to retrieve the model before deleting the record, you may use the `unsearchable` method on an Eloquent query instance:

```
Order::where('price', '>', 100)->unsearchable();
```

If you would like to remove the search index records for all of the models in a relationship, you may invoke the `unsearchable` on the relationship instance:

```
$user->orders()->unsearchable();
```

Or, if you already have a collection of Eloquent models in memory, you may call the `unsearchable` method on the collection instance to remove the model instances from their corresponding index:

```
$orders->unsearchable();
```

Pausing Indexing

Sometimes you may need to perform a batch of Eloquent operations on a model without syncing the model data to your search index. You may do this using the `withoutSyncingToSearch` method. This method accepts a single closure which will be immediately executed. Any model operations that occur within the closure will not be synced to the model's index:

```
use App\Models\Order;

Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

Conditionally Searchable Model Instances

Sometimes you may need to only make a model searchable under certain conditions. For example, imagine you have `App\Models\Post` model that may be in one of two states: "draft" and "published". You may only want to allow "published" posts to be searchable. To accomplish this, you may define a `shouldBeSearchable` method on your model:

```
/**
 * Determine if the model should be searchable.
 *
 * @return bool
 */
public function shouldBeSearchable()
{
    return $this->isPublished();
}
```

The `shouldBeSearchable` method is only applied when manipulating models through the `save` and `create` methods, queries, or relationships. Directly making models or collections searchable using the `searchable` method will override the result of the `shouldBeSearchable` method.

Searching

You may begin searching a model using the `search` method. The search method accepts a single string that will be used to search your models. You should then chain the `get` method onto the search query to retrieve the Eloquent models that match the given search query:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->get();
```

Since Scout searches return a collection of Eloquent models, you may even return the results directly from a route or controller and they will automatically be converted to JSON:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return Order::search($request->search)->get();
});
```

If you would like to get the raw search results before they are converted to Eloquent models, you may use the `raw` method:

```
$orders = Order::search('Star Trek')->raw();
```

Custom Indexes

Search queries will typically be performed on the index specified by the model's `searchableAs` method. However, you may use the `within` method to specify a custom index that should be searched instead:

```
$orders = Order::search('Star Trek')
    ->within('tv_shows_popularity_desc')
    ->get();
```

Where Clauses

Scout allows you to add simple "where" clauses to your search queries. Currently, these clauses only support basic numeric equality checks and are primarily useful for scoping search queries by an owner ID. Since a search index is not a relational database, more advanced "where" clauses are not currently supported:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->where('user_id', 1)->get();
```

Pagination

In addition to retrieving a collection of models, you may paginate your search results using the `paginate` method. This method will return an `Illuminate\Pagination\LengthAwarePaginator` instance just as if you had paginated a traditional Eloquent query:

```
use App\Models\Order;

$orders = Order::search('Star Trek')->paginate();
```

You may specify how many models to retrieve per page by passing the amount as the first argument to the `paginate` method:

```
$orders = Order::search('Star Trek')->paginate(15);
```

Once you have retrieved the results, you may display the results and render the page links using Blade just as if you had paginated a traditional Eloquent query:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

Of course, if you would like to retrieve the pagination results as JSON, you may return the paginator instance directly from a route or controller:

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    return Order::search($request->input('query'))->paginate(15);
});
```

Soft Deleting

If your indexed models are soft deleting and you need to search your soft deleted models, set the `soft_delete` option of the `config/scout.php` configuration file to `true`:

```
'soft_delete' => true,
```

When this configuration option is `true`, Scout will not remove soft deleted models from the search index. Instead, it will set a hidden `__soft_deleted` attribute on the indexed record. Then, you may use the `withTrashed` or `onlyTrashed` methods to retrieve the soft deleted records when searching:

```
use App\Models\Order;

// Include trashed records when retrieving results...
$orders = Order::search('Star Trek')->withTrashed()->get();

// Only include trashed records when retrieving results...
$orders = Order::search('Star Trek')->onlyTrashed()->get();
```

{tip} When a soft deleted model is permanently deleted using `forceDelete`, Scout will remove it from the search index automatically.

Customizing Engine Searches

If you need to perform advanced customization of the search behavior of an engine you may pass a closure as the second argument to the `search` method. For example, you could use this callback to add geo-location data to your search options before the search query is passed to Algolia:

```
use Algolia\AlgoliaSearch\SearchIndex;
use App\Models\Order;

Order::search(
    'Star Trek',
    function (SearchIndex $algolia, string $query, array $options) {
        $options['body']['query']['bool']['filter']['geo_distance'] = [
            'distance' => '1000km',
            'location' => ['lat' => 36, 'lon' => 111],
        ];

        return $algolia->search($query, $options);
    }
)->get();
```

Custom Engines

Writing The Engine

If one of the built-in Scout search engines doesn't fit your needs, you may write your own custom engine and register it with Scout. Your engine should extend the `Laravel\Scout\Engines\Engine` abstract class. This abstract class contains eight methods your custom engine must implement:

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

You may find it helpful to review the implementations of these methods on the `Laravel\Scout\Engines\AlgoliaEngine` class. This class will provide you with a good starting point for learning how to implement each of these methods in your own engine.

Registering The Engine

Once you have written your custom engine, you may register it with Scout using the `extend` method of the Scout engine manager. Scout's engine manager may be resolved from the Laravel service container. You should call the `extend` method from the `boot` method of your `App\Providers\AppServiceProvider` class or any other service provider used by your application:

```
use App\ScoutExtensions\MySqlSearchEngine
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

Once your engine has been registered, you may specify it as your default Scout `driver` in your application's `config/scout.php` configuration file:

```
'driver' => 'mysql',
```

Builder Macros

If you would like to define a custom Scout search builder method, you may use the `macro` method on the `Laravel\Scout\Builder` class. Typically, "macros" should be defined within a service provider's `boot` method:

```
use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;
use Laravel\Scout\Builder;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Builder::macro('count', function () {
        return $this->engine()->getTotalCount(
            $this->engine()->search($this)
        );
    });
}
```

The `macro` function accepts a macro name as its first argument and a closure as its second argument. The macro's closure will be executed when calling the macro name from a `Laravel\Scout\Builder` implementation:

```
use App\Models\Order;

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

Database: Seeding

Introduction

Laravel includes the ability to seed your database with data using seed classes. All seed classes are stored in the `database/seeders` directory. By default, a `DatabaseSeeder` class is defined for you. From this class, you may use the `call` method to run other seed classes, allowing you to control the seeding order.

{tip} Mass assignment protection is automatically disabled during database seeding.

Writing Seeders

To generate a seeder, execute the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeders` directory:

```
php artisan make:seeder UserSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the query builder to manually insert data or you may use Eloquent model factories.

As an example, let's modify the default `DatabaseSeeder` class and add a database insert statement to the `run` method:

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

{tip} You may type-hint any dependencies you need within the `run` method's signature. They will automatically be resolved via the Laravel service container.

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use model factories to conveniently generate large amounts of database records. First, review the model factory documentation to learn how to define your factories.

For example, let's create 50 users that each has one related post:

```
use App\Models\User;

/**
 * Run the database seeders.
 *
 * @return void
 */
public function run()
{
    User::factory()
            ->count(50)
            ->hasPosts(1)
            ->create();
}
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you may use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes too large. The `call` method accepts an array of seeder classes that should be executed:

```
/**
 * Run the database seeders.
 *
 * @return void
 */
public function run()
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

Muting Model Events

While running seeds, you may want to prevent models from dispatching events. You may achieve this using the `WithoutModelEvents` trait. When used, the `WithoutModelEvents` trait ensures no model events are dispatched, even if additional seed classes are executed via the `call` method:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;

class DatabaseSeeder extends Seeder
{
    use WithoutModelEvents;

    /**
     * Run the database seeders.
     *
     * @return void
     */
    public function run()
    {
        $this->call([
            UserSeeder::class,
        ]);
    }
}
```

Running Seeders

You may execute the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `Database\Seeders\DatabaseSeeder` class, which may in turn invoke other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed

php artisan db:seed --class=UserSeeder
```

You may also seed your database using the `migrate:fresh` command in combination with the `--seed` option, which will drop all tables and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:fresh --seed
```

Forcing Seeders To Run In Production

Some seeding operations may cause you to alter or lose data. In order to protect you from running seeding commands against your production database, you will be prompted for confirmation before the seeders are executed in the `production` environment. To force the seeders to run without a prompt, use the `--force` flag:

```
php artisan db:seed --force
```

Laravel Socialite

Introduction

In addition to typical, form based authentication, Laravel also provides a simple, convenient way to authenticate with OAuth providers using Laravel Socialite. Socialite currently supports authentication with Facebook, Twitter, LinkedIn, Google, GitHub, GitLab, and Bitbucket.

{tip} Adapters for other platforms are listed at the community driven Socialite Providers website.

Installation

To get started with Socialite, use the Composer package manager to add the package to your project's dependencies:

```
composer require laravel/socialite
```

Upgrading Socialite

When upgrading to a new major version of Socialite, it's important that you carefully review the upgrade guide.

Configuration

Before using Socialite, you will need to add credentials for the OAuth providers your application utilizes. These credentials should be placed in your application's `config/services.php` configuration file, and should use the key `facebook`, `twitter`, `linkedin`, `google`, `github`, `gitlab`, or `bitbucket`, depending on the providers your application requires:

```
'github' => [
    'client_id' => env('GITHUB_CLIENT_ID'),
    'client_secret' => env('GITHUB_CLIENT_SECRET'),
    'redirect' => 'http://example.com/callback-url',
],
```

{tip} If the `redirect` option contains a relative path, it will automatically be resolved to a fully qualified URL.

Authentication

Routing

To authenticate users using an OAuth provider, you will need two routes: one for redirecting the user to the OAuth provider, and another for receiving the callback from the provider after authentication. The example controller below demonstrates the implementation of both routes:

```
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/redirect', function () {
    return Socialite::driver('github')->redirect();
});

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // $user->token
});
```

The `redirect` method provided by the `Socialite` facade takes care of redirecting the user to the OAuth provider, while the `user` method will read the incoming request and retrieve the user's information from the provider after they are authenticated.

Authentication & Storage

Once the user has been retrieved from the OAuth provider, you may determine if the user exists in your application's database and authenticate the user. If the user does not exist in your application's database, you will typically create a new record in your database to represent the user:

```
use App\Models\User;
use Illuminate\Support\Facades\Auth;
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/callback', function () {
    $githubUser = Socialite::driver('github')->user();

    $user = User::where('github_id', $githubUser->id)->first();

    if ($user) {
        $user->update([
            'github_token' => $githubUser->token,
            'github_refresh_token' => $githubUser->refreshToken,
        ]);
    } else {
        $user = User::create([
            'name' => $githubUser->name,
            'email' => $githubUser->email,
            'github_id' => $githubUser->id,
            'github_token' => $githubUser->token,
            'github_refresh_token' => $githubUser->refreshToken,
        ]);
    }

    Auth::login($user);

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

{tip} For more information regarding what user information is available from specific OAuth providers, please consult the documentation on retrieving user details.

Access Scopes

Before redirecting the user, you may also add additional "scopes" to the authentication request using the `scopes` method. This method will merge all existing scopes with the scopes that you supply:

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();
```

You can overwrite all existing scopes on the authentication request using the `setScopes` method:

```
return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();
```

Optional Parameters

A number of OAuth providers support optional parameters in the redirect request. To include any optional parameters in the request, call the `with` method with an associative array:

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```

{note} When using the `with` method, be careful not to pass any reserved keywords such as `state` or `response_type`.

Retrieving User Details

After the user is redirected back to your authentication callback route, you may retrieve the user's details using Socialite's `user` method. The user object returned by the `user` method provides a variety of properties and methods you may use to store information about the user in your own database. Different properties and methods may be available depending on whether the OAuth provider you are authenticating with supports OAuth 1.0 or OAuth 2.0:

```
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // OAuth 2.0 providers...
    $token = $user->token;
    $refreshToken = $user->refreshToken;
    $expiresIn = $user->expiresIn;

    // OAuth 1.0 providers...
    $token = $user->token;
    $tokenSecret = $user->tokenSecret;

    // All providers...
    $user->getId();
    $user->getNickname();
    $user->getName();
    $user->getEmail();
    $user->getAvatar();
});
```

Retrieving User Details From A Token (OAuth2)

If you already have a valid access token for a user, you can retrieve their details using Socialite's `userFromToken` method:

```
use Laravel\Socialite\Facades\Socialite;

$user = Socialite::driver('github')->userFromToken($token);
```

Retrieving User Details From A Token And Secret (OAuth1)

If you already have a valid token and secret for a user, you can retrieve their details using Socialite's `userFromTokenAndSecret` method:

```
use Laravel\Socialite\Facades\Socialite;

$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);
```

Stateless Authentication

The `stateless` method may be used to disable session state verification. This is useful when adding social authentication to an API:

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('google')->stateless()->user();
```

{note} Stateless authentication is not available for the Twitter driver, which uses OAuth 1.0 for authentication.

Starter Kits

Introduction

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

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

Laravel Breeze

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's default view layer is made up of simple Blade templates styled with Tailwind CSS.

Breeze provides a wonderful starting point for beginning a fresh Laravel application and is also great choice for projects that plan to take their Blade templates to the next level with Laravel Livewire.

Installation

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

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

cd example-app

php artisan migrate
```

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

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

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

```
php artisan breeze:install

npm install
npm run dev
php artisan migrate
```

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

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

Breeze & Inertia

Laravel Breeze also offers an Inertia.js frontend implementation powered by Vue or React. To use an Inertia stack, specify `vue` or `react` as your desired stack when executing the `breeze:install` Artisan command:

```
php artisan breeze:install vue

// Or...

php artisan breeze:install react

npm install
npm run dev
php artisan migrate
```

Breeze & Next.js / API

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

```
php artisan breeze:install api

php artisan migrate
```

During installation, Breeze will add a `FRONTEND_URL` environment variable to your application's `.env` file. This URL should be the URL of your JavaScript application. This will typically be `http://localhost:3000` during local development.

Next.js Reference Implementation

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

Laravel Jetstream

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

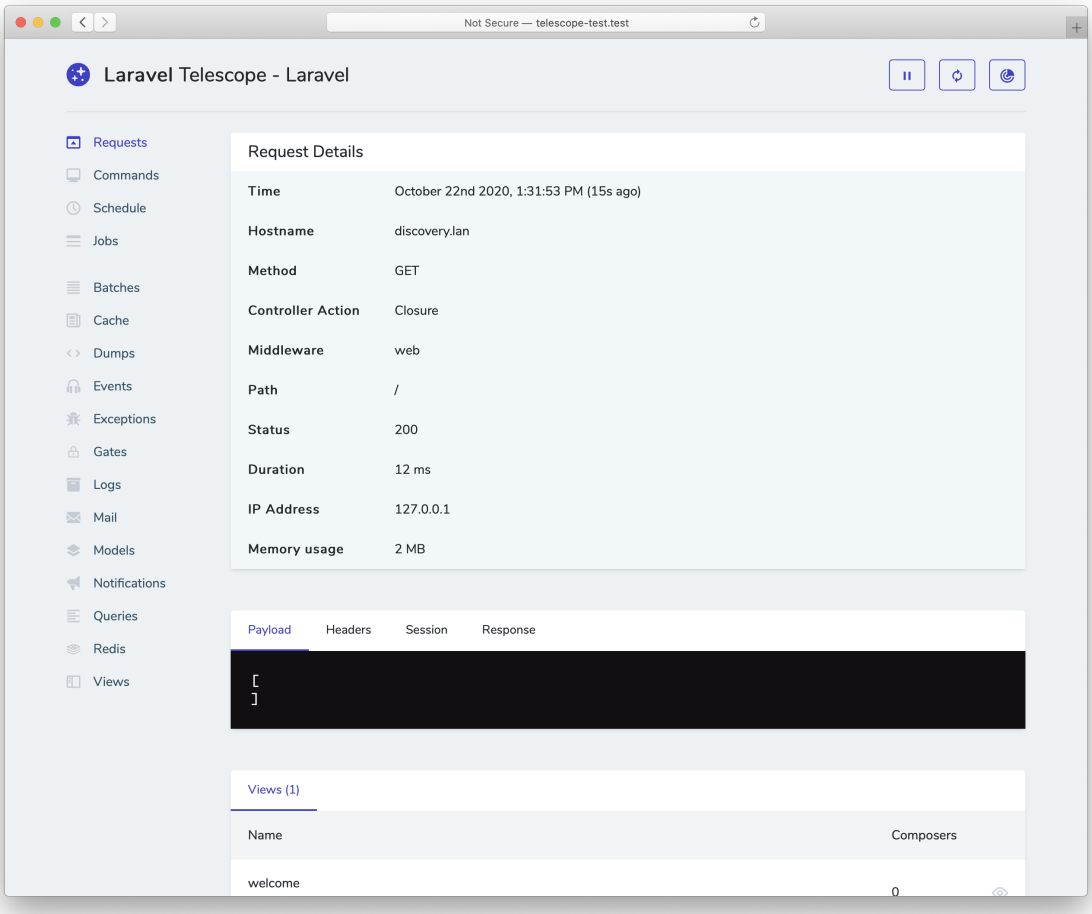
Jetstream provides a beautifully designed application scaffolding for Laravel and includes login, registration, email verification, two-factor authentication, session management, API support via Laravel Sanctum, and optional team management. Jetstream is designed using Tailwind CSS and offers your choice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

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

Laravel Telescope

Introduction

Telescope makes a wonderful companion to your local Laravel development environment. Telescope provides insight into the requests coming into your application, exceptions, log entries, database queries, queued jobs, mail, notifications, cache operations, scheduled tasks, variable dumps, and more.



Installation

You may use the Composer package manager to install Telescope into your Laravel project:

```
composer require laravel/telescope
```

After installing Telescope, publish its assets using the `telescope:install` Artisan command. After installing Telescope, you should also run the `migrate` command in order to create the tables needed to store Telescope's data:

```
php artisan telescope:install

php artisan migrate
```

Migration Customization

If you are not going to use Telescope's default migrations, you should call the `Telescope::ignoreMigrations` method in the `register` method of your application's `App\Providers\AppServiceProvider` class. You may export the default migrations using the following command: `php artisan vendor:publish --tag=telescope-migrations`

Local Only Installation

If you plan to only use Telescope to assist your local development, you may install Telescope using the `--dev` flag:

```
composer require laravel/telescope --dev

php artisan telescope:install

php artisan migrate
```

After running `telescope:install`, you should remove the

`TelescopeServiceProvider` service provider registration from your application's `config/app.php` configuration file. Instead, manually register Telescope's service providers in the `register` method of your `App\Providers\AppServiceProvider` class. We will ensure the current environment is `local` before registering the providers:

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    if ($this->app->environment('local')) {
$this->app->register(\Laravel\Telescope\TelescopeServiceProvider::class)
;
        $this->app->register(TelescopeServiceProvider::class);
    }
}
```

Finally, you should also prevent the Telescope package from being auto-discovered by adding the following to your `composer.json` file:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "laravel/telescope"
        ]
    }
},
```

Configuration

After publishing Telescope's assets, its primary configuration file will be located at `config/telescope.php`. This configuration file allows you to configure your watcher options. Each configuration option includes a description of its purpose, so be sure to thoroughly explore this file.

If desired, you may disable Telescope's data collection entirely using the `enabled` configuration option:

```
'enabled' => env('TELESCOPE_ENABLED', true),
```

Data Pruning

Without pruning, the `telescope_entries` table can accumulate records very quickly. To mitigate this, you should schedule the `telescope:prune` Artisan command to run daily:

```
$schedule->command('telescope:prune')->daily();
```

By default, all entries older than 24 hours will be pruned. You may use the `hours` option when calling the command to determine how long to retain Telescope data. For example, the following command will delete all records created over 48 hours ago:

```
$schedule->command('telescope:prune --hours=48')->daily();
```

Dashboard Authorization

The Telescope dashboard may be accessed at the `/telescope` route. By default, you will only be able to access this dashboard in the `local` environment. Within your `app/Providers/TelescopeServiceProvider.php` file, there is an authorization gate definition. This authorization gate controls access to Telescope in **non-local** environments. You are free to modify this gate as needed to restrict access to your Telescope installation:

```
/**
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope in non-local
environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

{note} You should ensure you change your `APP_ENV` environment variable to `production` in your production environment. Otherwise, your Telescope installation will be publicly available.

Upgrading Telescope

When upgrading to a new major version of Telescope, it's important that you carefully review the upgrade guide.

In addition, when upgrading to any new Telescope version, you should re-publish Telescope's assets:

```
php artisan telescope:publish
```

To keep the assets up-to-date and avoid issues in future updates, you may add the `telescope:publish` command to the `post-update-cmd` scripts in your application's `composer.json` file:

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan telescope:publish --ansi"
        ]
    }
}
```

Filtering

Entries

You may filter the data that is recorded by Telescope via the `filter` closure that is defined in your `App\Providers\TelescopeServiceProvider` class. By default, this closure records all data in the `local` environment and exceptions, failed jobs, scheduled tasks, and data with monitored tags in all other environments:

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filter(function (IncomingEntry $entry) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entry->isReportableException() ||
            $entry->isFailedJob() ||
            $entry->isScheduledTask() ||
            $entry->isSlowQuery() ||
            $entry->hasMonitoredTag();
    });
}
```

Batches

While the `filter` closure filters data for individual entries, you may use the `filterBatch` method to register a closure that filters all data for a given request or console command. If the closure returns `true`, all of the entries are recorded by Telescope:

```
use Illuminate\Support\Collection;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filterBatch(function (Collection $entries) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entries->contains(function ($entry) {
            return $entry->isReportableException() ||
                $entry->isFailedJob() ||
                $entry->isScheduledTask() ||
                $entry->isSlowQuery() ||
                $entry->hasMonitoredTag();
            });
    });
}
```

Tagging

Telescope allows you to search entries by "tag". Often, tags are Eloquent model class names or authenticated user IDs which Telescope automatically adds to entries. Occasionally, you may want to attach your own custom tags to entries. To accomplish this, you may use the `Telescope::tag` method. The `tag` method accepts a closure which should return an array of tags. The tags returned by the closure will be merged with any tags Telescope would automatically attach to the entry. Typically, you should call the `tag` method within the `register` method of your `App\Providers\TelescopeServiceProvider` class:

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::tag(function (IncomingEntry $entry) {
        return $entry->type === 'request'
                    ? ['status:'.$entry->content['response_status']]
                    : [];
    });
 }
```

Available Watchers

Telescope "watchers" gather application data when a request or console command is executed. You may customize the list of watchers that you would like to enable within your `config/telescope.php` configuration file:

```
'watchers' => [
    Watchers\CacheWatcher::class => true,
    Watchers\CommandWatcher::class => true,
    ...
],
```

Some watchers also allow you to provide additional customization options:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 100,
    ],
    ...
],
```

Batch Watcher

The batch watcher records information about queued batches, including the job and connection information.

Cache Watcher

The cache watcher records data when a cache key is hit, missed, updated and forgotten.

Command Watcher

The command watcher records the arguments, options, exit code, and output whenever an Artisan command is executed. If you would like to exclude certain commands from being recorded by the watcher, you may specify the command in the `ignore` option within your `config/telescope.php` file:

```
'watchers' => [
    Watchers\CommandWatcher::class => [
        'enabled' => env('TELESCOPE_COMMAND_WATCHER', true),
        'ignore' => ['key:generate'],
    ],
    ...
],
```

Dump Watcher

The dump watcher records and displays your variable dumps in Telescope. When using Laravel, variables may be dumped using the global `dump` function. The dump watcher tab must be open in a browser for the dump to be recorded, otherwise, the dumps will be ignored by the watcher.

Event Watcher

The event watcher records the payload, listeners, and broadcast data for any events dispatched by your application. The Laravel framework's internal events are ignored by the Event watcher.

Exception Watcher

The exception watcher records the data and stack trace for any reportable exceptions that are thrown by your application.

Gate Watcher

The gate watcher records the data and result of gate and policy checks by your application. If you would like to exclude certain abilities from being recorded by the watcher, you may specify those in the `ignore_abilities` option in your `config/telescope.php` file:

```
'watchers' => [
    Watchers\GateWatcher::class => [
        'enabled' => env('TELESCOPE_GATE_WATCHER', true),
        'ignore_abilities' => ['viewNova'],
    ],
    ...
],
```

HTTP Client Watcher

The HTTP client watcher records outgoing HTTP client requests made by your application.

Job Watcher

The job watcher records the data and status of any jobs dispatched by your application.

Log Watcher

The log watcher records the log data for any logs written by your application.

Mail Watcher

The mail watcher allows you to view an in-browser preview of emails sent by your application along with their associated data. You may also download the email as an `.eml` file.

Model Watcher

The model watcher records model changes whenever an Eloquent model event is dispatched. You may specify which model events should be recorded via the watcher's `events` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
    ],
    ...
],
```

If you would like to record the number of models hydrated during a given request, enable the `hydrations` option:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
        'hydrations' => true,
    ],
    ...
],
```

Notification Watcher

The notification watcher records all notifications sent by your application. If the notification triggers an email and you have the mail watcher enabled, the email will also be available for preview on the mail watcher screen.

Query Watcher

The query watcher records the raw SQL, bindings, and execution time for all queries that are executed by your application. The watcher also tags any queries slower than 100

milliseconds as `slow`. You may customize the slow query threshold using the watcher's `slow` option:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 50,
    ],
    ...
],
```

Redis Watcher

The Redis watcher records all Redis commands executed by your application. If you are using Redis for caching, cache commands will also be recorded by the Redis watcher.

Request Watcher

The request watcher records the request, headers, session, and response data associated with any requests handled by the application. You may limit your recorded response data via the `size_limit` (in kilobytes) option:

```
'watchers' => [
    Watchers\RequestWatcher::class => [
        'enabled' => env('TELESCOPE_REQUEST_WATCHER', true),
        'size_limit' => env('TELESCOPE_RESPONSE_SIZE_LIMIT', 64),
    ],
    ...
],
```

Schedule Watcher

The schedule watcher records the command and output of any scheduled tasks run by your application.

View Watcher

The view watcher records the view name, path, data, and "composers" used when rendering views.

Displaying User Avatars

The Telescope dashboard displays the user avatar for the user that was authenticated when a given entry was saved. By default, Telescope will retrieve avatars using the Gravatar web service. However, you may customize the avatar URL by registering a callback in your `App\Providers\TelescopeServiceProvider` class. The callback will receive the user's ID and email address and should return the user's avatar image URL:

```
use App\Models\User;
use Laravel\Telescope\Telescope;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    // ...

    Telescope::avatar(function ($id, $email) {
        return '/avatars/'.User::find($id)->avatar_path;
    });
}
```

Testing: Getting Started

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a `phpunit.xml` file is already set up for your application. The framework also ships with convenient helper methods that allow you to expressively test your applications.

By default, your application's `tests` directory contains two directories: `Feature` and `Unit`. Unit tests are tests that focus on a very small, isolated portion of your code. In fact, most unit tests probably focus on a single method. Tests within your "Unit" test directory do not boot your Laravel application and therefore are unable to access your application's database or other framework services.

Feature tests may test a larger portion of your code, including how several objects interact with each other or even a full HTTP request to a JSON endpoint. **Generally, most of your tests should be feature tests. These types of tests provide the most confidence that your system as a whole is functioning as intended.**

An `ExampleTest.php` file is provided in both the `Feature` and `Unit` test directories. After installing a new Laravel application, execute the `vendor/bin/phpunit` or `php artisan test` commands to run your tests.

Environment

When running tests, Laravel will automatically set the configuration environment to `testing` because of the environment variables defined in the `phpunit.xml` file. Laravel also automatically configures the session and cache to the `array` driver while testing, meaning no session or cache data will be persisted while testing.

You are free to define other testing environment configuration values as necessary. The `testing` environment variables may be configured in your application's `phpunit.xml` file, but make sure to clear your configuration cache using the `config:clear` Artisan command before running your tests!

The `.env.testing` Environment File

In addition, you may create a `.env.testing` file in the root of your project. This file will be used instead of the `.env` file when running PHPUnit tests or executing Artisan commands with the `--env=testing` option.

The `CreatesApplication` Trait

Laravel includes a `CreatesApplication` trait that is applied to your application's base `TestCase` class. This trait contains a `createApplication` method that bootstraps the Laravel application before running your tests. It's important that you leave this trait at its original location as some features, such as Laravel's parallel testing feature, depend on it.

Creating Tests

To create a new test case, use the `make:test` Artisan command. By default, tests will be placed in the `tests/Feature` directory:

```
php artisan make:test UserTest
```

If you would like to create a test within the `tests/Unit` directory, you may use the `--unit` option when executing the `make:test` command:

```
php artisan make:test UserTest --unit
```

If you would like to create a Pest PHP test, you may provide the `--pest` option to the `make:test` command:

```
php artisan make:test UserTest --pest
php artisan make:test UserTest --unit --pest
```

{tip} Test stubs may be customized using stub publishing.

Once the test has been generated, you may define test methods as you normally would using PHPUnit. To run your tests, execute the `vendor/bin/phpunit` or `php artisan test` command from your terminal:

```php
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function test_basic_test()
    {
        $this->assertTrue(true);
    }
}
```

{note} If you define your own `setUp` / `tearDown` methods within a test class, be sure to call the respective `parent::setUp()` / `parent::tearDown()` methods on the parent class.

Running Tests

As mentioned previously, once you've written tests, you may run them using `phpunit`:

```
./vendor/bin/phpunit
```

In addition to the `phpunit` command, you may use the `test` Artisan command to run your tests. The Artisan test runner provides verbose test reports in order to ease development and debugging:

```
php artisan test
```

Any arguments that can be passed to the `phpunit` command may also be passed to the Artisan `test` command:

```
php artisan test --testsuite=Feature --stop-on-failure
```

Running Tests In Parallel

By default, Laravel and PHPUnit execute your tests sequentially within a single process. However, you may greatly reduce the amount of time it takes to run your tests by running tests simultaneously across multiple processes. To get started, ensure your application depends on version `^5.3` or greater of the `nunomaduro/collision` package. Then, include the `--parallel` option when executing the `test` Artisan command:

```
php artisan test --parallel
```

By default, Laravel will create as many processes as there are available CPU cores on your machine. However, you may adjust the number of processes using the `--processes` option:

```
php artisan test --parallel --processes=4
```

{note} When running tests in parallel, some PHPUnit options (such as `--do-not-cache-result`) may not be available.

Parallel Testing & Databases

Laravel automatically handles creating and migrating a test database for each parallel process that is running your tests. The test databases will be suffixed with a process token which is unique per process. For example, if you have two parallel test processes, Laravel will create and use `your_db_test_1` and `your_db_test_2` test databases.

By default, test databases persist between calls to the `test` Artisan command so that they can be used again by subsequent `test` invocations. However, you may re-create them using the `--recreate-databases` option:

```
php artisan test --parallel --recreate-databases
```

Parallel Testing Hooks

Occasionally, you may need to prepare certain resources used by your application's tests so they may be safely used by multiple test processes.

Using the `ParallelTesting` facade, you may specify code to be executed on the `setUp` and `tearDown` of a process or test case. The given closures receive the `$token` and `$testCase` variables that contain the process token and the current test case, respectively:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\ParallelTesting;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        ParallelTesting::setUpProcess(function ($token) {
            // ...
        });

        ParallelTesting::setUpTestCase(function ($token, $testCase) {
            // ...
        });

        // Executed when a test database is created...
        ParallelTesting::setUpTestDatabase(function ($database, $token)
{
            Artisan::call('db:seed');
        });

        ParallelTesting::tearDownTestCase(function ($token, $testCase) {
            // ...
        });

        ParallelTesting::tearDownProcess(function ($token) {
            // ...
        });
    }
}
```

Accessing The Parallel Testing Token

If you would like to access to current parallel process "token" from any other location in your application's test code, you may use the `token` method. This token is a unique, string identifier for an individual test process and may be used to segment resources across parallel test processes. For example, Laravel automatically appends this token to the end of the test databases created by each parallel testing process:

```
$token = ParallelTesting::token();
```

Upgrade Guide

Medium Impact Changes

- [Belongs To Many `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods](#belongs-to-many-first-or-new) - [Postgres "Schema" Configuration](#postgres-schema-configuration) - [The `when` / `unless` Methods](#when-and-unless-methods) - [Custom Casts & `null`](#custom-casts-and-null) - [The `password` Rule](#the-password-rule) - [The `lang` Directory](#the-lang-directory) - [The `assertDeleted` Method](#the-assert-deleted-method)

Upgrading To 9.0 From 8.x

Estimated Upgrade Time: 15 Minutes

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

Application

The `Application` Contract

Likelihood Of Impact: Low

The `storagePath` method of the `Illuminate\Contracts\Foundation\Application` interface has been updated to accept a `$path` argument. If you are implementing this interface you should update your implementation accordingly:

```
public function storagePath($path = '');
```

Exception Handler `ignore` Method

Likelihood Of Impact: Low

The exception handler's `ignore` method is now `public` instead of `protected`. This method is not included in the default application skeleton; however, if you have manually defined this method you should update its visibility to `public`:

```
public function ignore(string $class);
```

Blade

Lazy Collections & The `$loop` Variable

Likelihood Of Impact: Low

When iterating over a `LazyCollection` instance within a Blade template, the `$loop` variable is no longer available, as accessing this variable causes the entire `LazyCollection` to be loaded into memory, thus rendering the usage of lazy collections pointless in this scenario.

Container

The `Container` Contract

Likelihood Of Impact: Very Low

The `Illuminate\Contracts\Container\Container` contract has received two method definitions: `scoped` and `scopedIf`. If you are manually implementing this contract, you should update your implementation to reflect these new methods.

The `ContextualBindingBuilder` Contract

Likelihood Of Impact: Very Low

The `Illuminate\Contracts\Container\ContextualBindingBuilder` contract now defines a `giveConfig` method. If you are manually implementing this interface, you should update your implementation to reflect this new method:

```
public function giveConfig($key, $default = null);
```

Database

Postgres "Schema" Configuration

Likelihood Of Impact: Medium

The `schema` configuration option used to configure Postgres connection search paths in your application's `config/database.php` configuration file should be renamed to `search_path`.

Eloquent

Custom Casts & `null`

Likelihood Of Impact: Medium

In previous releases of Laravel, the `set` method of custom cast classes was not invoked if the cast attribute was being set to `null`. However, this behavior was inconsistent with the Laravel documentation. In Laravel 9, the `set` method of the cast class will be invoked with `null` as the provided `$value` argument. Therefore, you should ensure your custom casts are able to sufficiently handle this scenario:

```
/**
 * Prepare the given value for storage.
 *
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @param  string  $key
 * @param  AddressModel  $value
 * @param  array  $attributes
 * @return array
 */
public function set($model, $key, $value, $attributes)
{
    if (! $value instanceof AddressModel) {
        throw new InvalidArgumentException('The given value is not an
Address instance.');
    }

    return [
        'address_line_one' => $value->lineOne,
        'address_line_two' => $value->lineTwo,
    ];
}
```

Belongs To Many `firstOrNew`, `firstOrCreate`, and `updateOrCreate` Methods

Likelihood Of Impact: Medium

The `belongsToMany` relationship's `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods all accept an array of attributes as their first argument. In previous releases of Laravel, this array of attributes was compared against the "pivot" / intermediate table for

existing records.

However, this behavior was unexpected and typically unwanted. Instead, these methods now compare the array of attributes against the table of the related model:

```
$user->roles()->updateOrCreate([
    'name' => 'Administrator',
]);
```

In addition, the `firstOrCreate` method now accepts a `$values` array as its second argument. This array will be merged with the first argument to the method (`$attributes`) when creating the related model if one does not already exist. This changes makes this method consistent with the `firstOrCreate` methods offered by other relationship types:

```
$user->roles()->firstOrCreate([
    'name' => 'Administrator',
], [
    'created_by' => $user->id,
]);
```

The `touch` Method

Likelihood Of Impact: Low

The `touch` method now accepts an attribute to touch. If you were previously overwriting this method, you should update your method signature to reflect this new argument:

```
public function touch($attribute = null);
```

Encryption

The Encrypter Contract

Likelihood Of Impact: Low

The `Illuminate\Contracts\Encryption\Encrypter` contract now defines a `getKey` method. If you are manually implementing this interface, you should update your implementation accordingly:

```
public function getKey();
```

Facades

The `getFacadeAccessor` Method

Likelihood Of Impact: Low

The `getFacadeAccessor` method must always return a container binding key. In previous releases of Laravel, this method could return an object instance; however, this behavior is no longer supported. If you have written your own facades, you should ensure that this method returns a container binding string:

```
/**
 * Get the registered name of the component.
 *
 * @return string
 */
protected static function getFacadeAccessor()
{
    return Example::class;
}
```

Filesystem

The `FILESYSTEM_DRIVER` Environment Variable

Likelihood Of Impact: Low

The `FILESYSTEM_DRIVER` environment variable has been renamed to `FILESYSTEM_DISK` to more accurately reflect its usage. This change only affects the application skeleton; however, you are welcome to update your own application's environment variables to reflect this change if you wish.

Helpers

The `data_get` Helper & Iterable Objects

Likelihood Of Impact: Very Low

Previously, the `data_get` helper could be used to retrieve nested data on arrays and `Collection` instances; however, this helper can now retrieve nested data on all iterable objects.

The `when` / `unless` Methods

Likelihood Of Impact: Medium

As you may know, `when` and `unless` methods are offered by various classes throughout the framework. These methods can be used to conditionally perform an action if the boolean value of the first argument to the method evaluates to `true` or `false`:

```
$collection->when(true, function ($collection) {
    $collection->merge([1, 2, 3]);
});
```

Therefore, in previous releases of Laravel, passing a closure to the `when` or `unless` methods meant that the conditional operation would always execute, since a loose comparison against a closure object (or any other object) always evaluates to `true`. This often led to unexpected outcomes because developers expect the **result** of the closure to be used as the boolean value that determines if the conditional action executes.

So, in Laravel 9, any closures passed to the `when` or `unless` methods will be executed and the value returned by the closure will be considered the boolean value used by the `when` and `unless` methods:

```
$collection->when(function ($collection) {
    // This closure is executed...
    return false;
}, function ($collection) {
    // Not executed since first closure returned "false"...
    $collection->merge([1, 2, 3]);
});
```

Packages

The `lang` Directory

Likelihood Of Impact: Medium

In new Laravel applications, the `resources/lang` directory is now located in the root project directory (`lang`). If your package is publishing language files to this directory, you should ensure that your package is publishing to `app()->langPath()` instead of a hard-coded path.

Queue

The `opis/closure` Library

Likelihood Of Impact: Low

Laravel's dependency on `opis/closure` has been replaced by `laravel/serializable-closure`. This should not cause any breaking change in your application unless you are interacting with the `opis/closure` library directly. In addition, the previously deprecated `Illuminate\Queue\SerializableClosureFactory` and `Illuminate\Queue\SerializableClosure` classes have been removed. If you are interacting with `opis/closure` library directly or using any of the removed classes, you may use Laravel Serializable Closure instead.

The Failed Job Provider `failed` Method

Likelihood Of Impact: Low

The `flush` method defined by the `Illuminate\Queue\Failed\FailedJobProviderInterface` interface now accepts an `$age` argument which determines how old a failed job must be (in days) before it is flushed by the `queue:flush` command. If you are manually implementing the `FailedJobProviderInterface` you should ensure that your implementation is updated to reflect this new argument:

```
public function flush($age = null);
```

Testing

The `assertDeleted` Method

Likelihood Of Impact: Medium

All calls to the `assertDeleted` method should be updated to `assertModelMissing`.

Validation

Form Request `validated` Method

Likelihood Of Impact: Low

The `validated` method offered by form requests now accepts `$key` and `$default` arguments. If you are manually overwriting the definition of this method, you should update your method's signature to reflect these new arguments:

```
public function validated($key = null, $default = null)
```

The `password` Rule

Likelihood Of Impact: Medium

The `password` rule, which validates that the given input value matches the authenticated user's current password, has been renamed to `current_password`.

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.

Laravel Valet

Introduction

Valet is a Laravel development environment for macOS minimalists. Laravel Valet configures your Mac to always run Nginx in the background when your machine starts. Then, using DnsMasq, Valet proxies all requests on the `*.test` domain to point to sites installed on your local machine.

In other words, Valet is a blazing fast Laravel development environment that uses roughly 7 MB of RAM. Valet isn't a complete replacement for Sail or Homestead, but provides a great alternative if you want flexible basics, prefer extreme speed, or are working on a machine with a limited amount of RAM.

Out of the box, Valet support includes, but is not limited to:

- [Laravel](https://laravel.com) - [Lumen](https://lumen.laravel.com) - [Bedrock](https://roots.io/bedrock/) - [CakePHP 3](https://cakephp.org) - [Concrete5](https://www.concrete5.org/) - [Contao](https://contao.org/en/) - [Craft](https://craftcms.com) - [Drupal](https://www.drupal.org/) - [ExpressionEngine](https://www.expressionengine.com/) - [Jigsaw](https://jigsaw.tighten.co) - [Joomla](https://www.joomla.org/) - [Katana](https://github.com/themsaid/katana) - [Kirby](https://getkirby.com/) - [Magento](https://magento.com/) - [OctoberCMS](https://octobercms.com/) - [Sculpin](https://sculpin.io/) - [Slim](https://www.slimframework.com) - [Statamic](https://statamic.com) - Static HTML - [Symfony](https://symfony.com) - [WordPress](https://wordpress.org) - [Zend](https://framework.zend.com)

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

Installation

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

To get started, you first need to ensure that Homebrew is up to date using the `update` command:

```
brew update
```

Next, you should use Homebrew to install PHP:

```
brew install php
```

After installing PHP, you are ready to install the Composer package manager. In addition, you should make sure the `~/.composer/vendor/bin` directory is in your system's "PATH". After Composer has been installed, you may install Laravel Valet as a global Composer package:

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

Finally, you may execute Valet's `install` command. This will configure and install Valet and DnsMasq. In addition, the daemons Valet depends on will be configured to launch when your system starts:

```
valet install
```

Once Valet is installed, try pinging any `*.test` domain on your terminal using a command such as `ping foobar.test`. If Valet is installed correctly you should see this domain responding on `127.0.0.1`.

Valet will automatically start its required services each time your machine boots.

PHP Versions

Valet allows you to switch PHP versions using the `valet use php@version` command. Valet will install the specified PHP version via Homebrew if it is not already installed:

```
valet use php@7.2

valet use php
```

You may also create a `.valetphprc` file in the root of your project. The `.valetphprc` file should contain the PHP version the site should use:

```
php@7.2
```

Once this file has been created, you may simply execute the `valet use` command and the command will determine the site's preferred PHP version by reading the file.

{note} Valet only serves one PHP version at a time, even if you have multiple PHP versions installed.

Database

If your application needs a database, check out DBngin. DBngin provides a free, all-in-one database management tool that includes MySQL, PostgreSQL, and Redis. After DBngin has been installed, you can connect to your database at `127.0.0.1` using the `root` username and an empty string for the password.

Resetting Your Installation

If you are having trouble getting your Valet installation to run properly, executing the `composer global update` command followed by `valet install` will reset your installation and can solve a variety of problems. In rare cases, it may be necessary to "hard reset" Valet by executing `valet uninstall --force` followed by `valet install`.

Upgrading Valet

You may update your Valet installation by executing the `composer global update` command in your terminal. After upgrading, it is good practice to run the `valet install`

command so Valet can make additional upgrades to your configuration files if necessary.

Serving Sites

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

The `park` Command

The `park` command registers a directory on your machine that contains your applications. Once the directory has been "parked" with Valet, all of the directories within that directory will be accessible in your web browser at `http://<directory-name>.test`:

```
cd ~/Sites

valet park
```

That's all there is to it. Now, any application you create within your "parked" directory will automatically be served using the `http://<directory-name>.test` convention. So, if your parked directory contains a directory named "laravel", the application within that directory will be accessible at `http://laravel.test`. In addition, Valet automatically allows you to access the site using wildcard subdomains (`http://foo.laravel.test`).

The `link` Command

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

```
cd ~/Sites/laravel

valet link
```

Once an application has been linked to Valet using the `link` command, you may access the application using its directory name. So, the site that was linked in the example above may be accessed at `http://laravel.test`. In addition, Valet automatically allows you to access the site using wildcard sub-domains (`http://foo.laravel.test`).

If you would like to serve the application at a different hostname, you may pass the hostname to the `link` command. For example, you may run the following command to make an application available at `http://application.test`:

```
cd ~/Sites/laravel

valet link application
```

You may execute the `links` command to display a list of all of your linked directories:

```
valet links
```

The `unlink` command may be used to destroy the symbolic link for a site:

```
cd ~/Sites/laravel

valet unlink
```

Securing Sites With TLS

By default, Valet serves sites over HTTP. However, if you would like to serve a site over encrypted TLS using HTTP/2, you may use the `secure` command. For example, if your site is being served by Valet on the `laravel.test` domain, you should run the following command to secure it:

```
valet secure laravel
```

To "unsecure" a site and revert back to serving its traffic over plain HTTP, use the `unsecure` command. Like the `secure` command, this command accepts the hostname that you wish to unsecure:

```
valet unsecure laravel
```

Serving A Default Site

Sometimes, you may wish to configure Valet to serve a "default" site instead of a `404` when visiting an unknown `test` domain. To accomplish this, you may add a `default` option to your `~/.config/valet/config.json` configuration file containing the path to the site that should serve as your default site:

```
"default": "/Users/Sally/Sites/foo",
```

Sharing Sites

Valet even includes a command to share your local sites with the world, providing an easy way to test your site on mobile devices or share it with team members and clients.

Sharing Sites Via Ngrok

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

```
cd ~/Sites/laravel

valet share
```

To stop sharing your site, you may press `Control + C`.

{tip} You may pass additional Ngrok parameters to the share command, such as `valet share --region=eu`. For more information, consult the ngrok documentation.

Sharing Sites Via Expose

If you have Expose installed, you can share your site by navigating to the site's directory in your terminal and running the `expose` command. Consult the Expose documentation for information regarding the additional command-line parameters it supports. After sharing the site, Expose will display the sharable URL that you may use on your other devices or amongst team members:

```
cd ~/Sites/laravel

expose
```

To stop sharing your site, you may press `Control + C`.

Sharing Sites On Your Local Network

Valet restricts incoming traffic to the internal `127.0.0.1` interface by default so that your development machine isn't exposed to security risks from the Internet.

If you wish to allow other devices on your local network to access the Valet sites on your machine via your machine's IP address (eg: `192.168.1.10/application.test`), you will need to manually edit the appropriate Nginx configuration file for that site to remove the restriction on the `listen` directive. You should remove the `127.0.0.1:` prefix on the `listen` directive for ports 80 and 443.

If you have not run `valet secure` on the project, you can open up network access for all non-HTTPS sites by editing the `/usr/local/etc/nginx/valet/valet.conf` file. However, if you're serving the project site over HTTPS (you have run `valet secure` for the site) then you should edit the `~/.config/valet/Nginx/app-name.test` file.

Once you have updated your Nginx configuration, run the `valet restart` command to apply the configuration changes.

Site Specific Environment Variables

Some applications using other frameworks may depend on server environment variables but do not provide a way for those variables to be configured within your project. Valet allows you to configure site specific environment variables by adding a `.valet-env.php` file within the root of your project. This file should return an array of site / environment variable pairs which will be added to the global `$_SERVER` array for each site specified in the array:

```
<?php

return [
    // Set $_SERVER['key'] to "value" for the laravel.test site...
    'laravel' => [
        'key' => 'value',
    ],

    // Set $_SERVER['key'] to "value" for all sites...
    '*' => [
        'key' => 'value',
    ],
];
```

Proxying Services

Sometimes you may wish to proxy a Valet domain to another service on your local machine. For example, you may occasionally need to run Valet while also running a separate site in Docker; however, Valet and Docker can't both bind to port 80 at the same time.

To solve this, you may use the `proxy` command to generate a proxy. For example, you may proxy all traffic from `http://elasticsearch.test` to `http://127.0.0.1:9200`:

```
// Proxy over HTTP...
valet proxy elasticsearch http://127.0.0.1:9200

// Proxy over TLS + HTTP/2...
valet proxy elasticsearch http://127.0.0.1:9200 --secure
```

You may remove a proxy using the `unproxy` command:

```
valet unproxy elasticsearch
```

You may use the `proxies` command to list all site configurations that are proxied:

```
valet proxies
```

Custom Valet Drivers

You can write your own Valet "driver" to serve PHP applications running on a framework or CMS that is not natively supported by Valet. When you install Valet, a `~/.config/valet/Drivers` directory is created which contains a `SampleValetDriver.php` file. This file contains a sample driver implementation to demonstrate how to write a custom driver. Writing a driver only requires you to implement three methods: `serves`, `isStaticFile`, and `frontControllerPath`.

All three methods receive the `$sitePath`, `$siteName`, and `$uri` values as their arguments. The `$sitePath` is the fully qualified path to the site being served on your machine, such as `/Users/Lisa/Sites/my-project`. The `$siteName` is the "host" / "site name" portion of the domain (`my-project`). The `$uri` is the incoming request URI (`/foo/bar`).

Once you have completed your custom Valet driver, place it in the `~/.config/valet/Drivers` directory using the `FrameworkValetDriver.php` naming convention. For example, if you are writing a custom valet driver for WordPress, your filename should be `WordPressValetDriver.php`.

Let's take a look at a sample implementation of each method your custom Valet driver should implement.

The `serves` Method

The `serves` method should return `true` if your driver should handle the incoming request. Otherwise, the method should return `false`. So, within this method, you should attempt to determine if the given `$sitePath` contains a project of the type you are trying to serve.

For example, let's imagine we are writing a `WordPressValetDriver`. Our `serves` method might look something like this:

```
/**
 * Determine if the driver serves the request.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return bool
 */
public function serves($sitePath, $siteName, $uri)
{
    return is_dir($sitePath.'/wp-admin');
}
```

The `isStaticFile` Method

The `isStaticFile` should determine if the incoming request is for a file that is "static", such as an image or a stylesheet. If the file is static, the method should return the fully qualified path to the static file on disk. If the incoming request is not for a static file, the method should return `false`:

```
/**
 * Determine if the incoming request is for a static file.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return string|false
 */
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath.'/public/'.$uri)) {
        return $staticFilePath;
    }

    return false;
}
```

{note} The `isStaticFile` method will only be called if the `serves` method returns `true` for the incoming request and the request URI is not `/`.

The `frontControllerPath` Method

The `frontControllerPath` method should return the fully qualified path to your application's "front controller", which is typically an "index.php" file or equivalent:

```
/**
 * Get the fully resolved path to the application's front controller.
 *
 * @param  string  $sitePath
 * @param  string  $siteName
 * @param  string  $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath.'/public/index.php';
}
```

Local Drivers

If you would like to define a custom Valet driver for a single application, create a `LocalValetDriver.php` file in the application's root directory. Your custom driver may extend the base `ValetDriver` class or extend an existing application specific driver such as the `LaravelValetDriver`:

```
class LocalValetDriver extends LaravelValetDriver
{
    /**
     * Determine if the driver serves the request.
     *
     * @param  string  $sitePath
     * @param  string  $siteName
     * @param  string  $uri
     * @return bool
     */
    public function serves($sitePath, $siteName, $uri)
    {
        return true;
    }

    /**
     * Get the fully resolved path to the application's front
controller.
     *
     * @param  string  $sitePath
     * @param  string  $siteName
     * @param  string  $uri
     * @return string
     */
    public function frontControllerPath($sitePath, $siteName, $uri)
    {
        return $sitePath.'/public_html/index.php';
    }
}
```

Other Valet Commands

Command	Description
`valet forget`	Run this command from a "parked" directory to remove it from the parked directory list.
`valet log`	View a list of logs which are written by Valet's services.
`valet paths`	View all of your "parked" paths.
`valet restart`	Restart the Valet daemons.
`valet start`	Start the Valet daemons.
`valet stop`	Stop the Valet daemons.
`valet trust`	Add sudoers files for Brew and Valet to allow Valet commands to be run without prompting for your password.
`valet uninstall`	Uninstall Valet: shows instructions for manual uninstall. Pass the `--force` option to aggressively delete all of Valet's resources.

Valet Directories & Files

You may find the following directory and file information helpful while troubleshooting issues with your Valet environment:

`~/.config/valet`

Contains all of Valet's configuration. You may wish to maintain a backup of this directory.

`~/.config/valet/dnsmasq.d/`

This directory contains DNSMasq's configuration.

`~/.config/valet/Drivers/`

This directory contains Valet's drivers. Drivers determine how a particular framework / CMS is served.

`~/.config/valet/Extensions/`

This directory contains custom Valet extensions / commands.

`~/.config/valet/Nginx/`

This directory contains all of Valet's Nginx site configurations. These files are rebuilt when running the `install`, `secure`, and `tld` commands.

`~/.config/valet/Sites/`

This directory contains all of the symbolic links for your linked projects.

`~/.config/valet/config.json`

This file is Valet's master configuration file.

`~/.config/valet/valet.sock`

This file is the PHP-FPM socket used by Valet's Nginx installation. This will only exist if PHP is running properly.

`~/.config/valet/Log/fpm-php.www.log`

This file is the user log for PHP errors.

`~/.config/valet/Log/nginx-error.log`

This file is the user log for Nginx errors.

`/usr/local/var/log/php-fpm.log`

This file is the system log for PHP-FPM errors.

`/usr/local/var/log/nginx`

This directory contains the Nginx access and error logs.

`/usr/local/etc/php/X.X/conf.d`

This directory contains the `*.ini` files for various PHP configuration settings.

`/usr/local/etc/php/X.X/php-fpm.d/valet-fpm.conf`

This file is the PHP-FPM pool configuration file.

`~/.composer/vendor/laravel/valet/cli/stubs/secure.valet.conf`

This file is the default Nginx configuration used for building SSL certificates for your sites.

Email Verification

Introduction

Many web applications require users to verify their email addresses before using the application. Rather than forcing you to re-implement this feature by hand for each application you create, Laravel provides convenient built-in services for sending and verifying email verification requests.

{tip} Want to get started fast? Install one of the Laravel application starter kits in a fresh Laravel application. The starter kits will take care of scaffolding your entire authentication system, including email verification support.

Model Preparation

Before getting started, verify that your `App\Models\User` model implements the `Illuminate\Contracts\Auth\MustVerifyEmail` contract:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

Once this interface has been added to your model, newly registered users will automatically be sent an email containing an email verification link. As you can see by examining your application's `App\Providers\EventServiceProvider`, Laravel already contains a `SendEmailVerificationNotification` listener that is attached to the `Illuminate\Auth\Events\Registered` event. This event listener will send the email verification link to the user.

If you are manually implementing registration within your application instead of using a

starter kit, you should ensure that you are dispatching the `Illuminate\Auth\Events\Registered` event after a user's registration is successful:

```
use Illuminate\Auth\Events\Registered;

event(new Registered($user));
```

Database Preparation

Next, your `users` table must contain an `email_verified_at` column to store the date and time that the user's email address was verified. By default, the `users` table migration included with the Laravel framework already includes this column. So, all you need to do is run your database migrations:

```
php artisan migrate
```

Routing

To properly implement email verification, three routes will need to be defined. First, a route will be needed to display a notice to the user that they should click the email verification link in the verification email that Laravel sent them after registration.

Second, a route will be needed to handle requests generated when the user clicks the email verification link in the email.

Third, a route will be needed to resend a verification link if the user accidentally loses the first verification link.

The Email Verification Notice

As mentioned previously, a route should be defined that will return a view instructing the user to click the email verification link that was emailed to them by Laravel after registration. This view will be displayed to users when they try to access other parts of the application without verifying their email address first. Remember, the link is automatically emailed to the user as long as your `App\Models\User` model implements the `MustVerifyEmail` interface:

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

The route that returns the email verification notice should be named `verification.notice`. It is important that the route is assigned this exact name since the `verified` middleware included with Laravel will automatically redirect to this route name if a user has not verified their email address.

{tip} When manually implementing email verification, you are required to define the contents of the verification notice view yourself. If you would like scaffolding that includes all necessary authentication and verification views, check out the Laravel application starter kits.

The Email Verification Handler

Next, we need to define a route that will handle requests generated when the user clicks the email verification link that was emailed to them. This route should be named `verification.verify` and be assigned the `auth` and `signed` middlewares:

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;

Route::get('/email/verify/{id}/{hash}', function
(EmailVerificationRequest $request) {
    $request->fulfill();

    return redirect('/home');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

Before moving on, let's take a closer look at this route. First, you'll notice we are using an `EmailVerificationRequest` request type instead of the typical `Illuminate\Http\Request` instance. The `EmailVerificationRequest` is a form request that is included with Laravel. This request will automatically take care of validating the request's `id` and `hash` parameters.

Next, we can proceed directly to calling the `fulfill` method on the request. This method will call the `markEmailAsVerified` method on the authenticated user and dispatch the `Illuminate\Auth\Events\Verified` event. The `markEmailAsVerified` method is available to the default `App\Models\User` model via the `Illuminate\Foundation\Auth\User` base class. Once the user's email address has been verified, you may redirect them wherever you wish.

Resending The Verification Email

Sometimes a user may misplace or accidentally delete the email address verification email. To accommodate this, you may wish to define a route to allow the user to request that the verification email be resent. You may then make a request to this route by placing a simple form submission button within your verification notice view:

```
use Illuminate\Http\Request;

Route::post('/email/verification-notification', function (Request
$request) {
    $request->user()->sendEmailVerificationNotification();

    return back()->with('message', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
```

Protecting Routes

Route middleware may be used to only allow verified users to access a given route. Laravel ships with a `verified` middleware, which references the `Illuminate\Auth\Middleware\EnsureEmailIsVerified` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('/profile', function () {
    // Only verified users may access this route...
})->middleware('verified');
```

If an unverified user attempts to access a route that has been assigned this middleware, they will automatically be redirected to the `verification.notice` named route.

Customization

Verification Email Customization

Although the default email verification notification should satisfy the requirements of most applications, Laravel allows you to customize how the email verification mail message is constructed.

To get started, pass a closure to the `toMailUsing` method provided by the `Illuminate\Auth\Notifications\VerifyEmail` notification. The closure will receive the notifiable model instance that is receiving the notification as well as the signed email verification URL that the user must visit to verify their email address. The closure should return an instance of `Illuminate\Notifications\Messages\MailMessage`. Typically, you should call the `toMailUsing` method from the `boot` method of your application's `App\Providers\AuthServiceProvider` class:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    // ...

    VerifyEmail::toMailUsing(function ($notifiable, $url) {
        return (new MailMessage)
            ->subject('Verify Email Address')
            ->line('Click the button below to verify your email
address.')
            ->action('Verify Email Address', $url);
    });
}
```

{tip} To learn more about mail notifications, please consult the mail notification documentation.

Events

When using the Laravel application starter kits, Laravel dispatches events during the email verification process. If you are manually handling email verification for your application, you may wish to manually dispatch these events after verification is completed. You may attach listeners to these events in your application's `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerifiedUser',
    ],
];
```