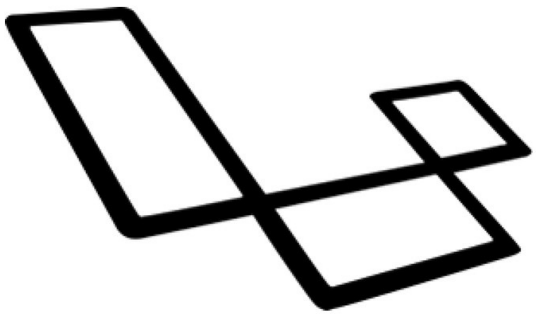


Laravel

5.1

DOCUMENTATION

Laravel Documentation - 5.1

https://laravel.com/docs/

eBook compiled from the source

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

by david@mundosaparte.com

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

Date: Thursday, 14-Jul-22 00:33:32 CEST

Contents

Prologue

Release Notes

Support Policy

For LTS releases, such as Laravel 5.1, bug fixes are provided for 2 years and security fixes are provided for 3 years. These releases provide the longest window of support and maintenance.

For general releases, bug fixes are provided for 6 months and security fixes are provided for 1 year.

Laravel 5.1.11

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

For more information, please refer to the authorization documentation.

Laravel 5.1.4

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

Laravel 5.1

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

PHP 5.5.9+

Since PHP 5.4 will enter "end of life" in September and will no longer receive security updates from the PHP development team, Laravel 5.1 requires PHP 5.5.9 or greater. PHP 5.5.9 allows compatibility with the latest versions of popular PHP libraries such as Guzzle and the AWS SDK.

LTS

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

PSR-2

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

Documentation

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

Event Broadcasting

In many modern web applications, web sockets are used to implement real-time, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a websocket connection to be handled by the client.

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

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

Middleware Parameters

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        return $next($request);
    }

}
```

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

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

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

Testing Overhaul

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

```
public function testNewUserRegistration()
{
    $this->visit('/register')
         ->type('Taylor', 'name')
         ->check('terms')
         ->press('Register')
```

```
        ->seePageIs('/dashboard');
}
```

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

Model Factories

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

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

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

Artisan Improvements

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

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

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

Folder Structure

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

Encryption

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

Laravel 5.0

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

New Folder Structure

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

Controllers, middleware, and requests (a new type of class in Laravel 5.0) are now grouped under the `app/Http`

directory, as they are all classes related to the HTTP transport layer of your application. Instead of a single, flat file of route filters, all middleware are now broken into their own class files.

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

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

Contracts

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

For more information on contracts, consult the full documentation.

Route Cache

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

Route Middleware

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

For more information on middleware, check out the documentation.

Controller Method Injection

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

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

Authentication Scaffolding

User registration, authentication, and password reset controllers are now included out of the box, as well as simple corresponding views, which are located at `resources/views/auth`. In addition, a "users" table migration has been included with the framework. Including these simple resources allows rapid development of application ideas without bogging down on authentication boilerplate. The authentication views may be accessed on the `auth/login` and `auth/register` routes. The `App\Services\Auth\Registrar` service is responsible for user validation and creation.

Event Objects

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

```
<?php

class PodcastWasPurchased
{
    public $podcast;
```

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

The event may be dispatched like normal:

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

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

```
<?php

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

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

Commands / Queueing

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

```
<?php

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

    protected $user, $podcast;

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

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

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

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

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

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

Database Queue

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

Laravel Scheduler

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

It looks like this:

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

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

Tinker / Psysh

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

```
php artisan tinker
```

DotEnv

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

Laravel Elixir

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

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

Laravel Socialite

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

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

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

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

Flysystem Integration

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

Amazon S3 is now as simple as:

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

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

Form Requests

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

```
<?php

namespace App\Http\Requests;

class RegisterRequest extends FormRequest
{
    public function rules()
    {
        return [
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:8',
        ];
    }

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

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

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

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

Simple Controller Request Validation

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

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

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

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

New Generators

To complement the new default application structure, new Artisan generator commands have been added to the

framework. See `php artisan list` for more details.

Configuration Cache

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

Symfony VarDumper

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

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

Laravel 4.2

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

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

PHP 5.4 Requirement

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

Laravel Forge

Laravel Forge, a new web based application, provides a simple way to create and manage PHP servers on the cloud of your choice, including Linode, DigitalOcean, Rackspace, and Amazon EC2. Supporting automated Nginx configuration, SSH key access, Cron job automation, server monitoring via NewRelic & Papertrail, "Push To Deploy", Laravel queue worker configuration, and more, Forge provides the simplest and most affordable way to launch all of your Laravel applications.

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

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

Laravel Homestead

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

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

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

Laravel Cashier

Laravel Cashier is a simple, expressive library for managing subscription billing with Stripe. With the introduction of Laravel 4.2, we are including Cashier documentation along with the main Laravel

documentation, though installation of the component itself is still optional. This release of Cashier brings numerous bug fixes, multi-currency support, and compatibility with the latest Stripe API.

Daemon Queue Workers

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

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

Mail API Drivers

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

Soft Deleting Traits

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

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

Convenient Auth & Remindable Traits

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

"Simple Paginate"

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

Migration Confirmation

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

Laravel 4.1

Full Change List

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

New SSH Component

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

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

Boris In Tinker

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

Eloquent Improvements

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

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

Database Read / Write Connections

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

Queue Priority

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

Failed Queue Job Handling

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

Cache Tags

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

Flexible Password Reminders

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

Improved Routing Engine

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

Improved Session Engine

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

Doctrine DBAL

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

Prologue

Upgrade Guide

Upgrading To 5.1.11

Laravel 5.1.11 includes support for authorization and policies. Incorporating these new features into your existing Laravel 5.1 applications is simple.

> **Note:** These upgrades are **optional**, and ignoring them will not affect your application.

Create The Policies Directory

First, create an empty `app/Policies` directory within your application.

Create / Register The AuthServiceProvider & Gate Facade

Create a `AuthServiceProvider` within your `app/Providers` directory. You may copy the contents of the default provider from GitHub. Remember to change the provider's namespace if your application is using a custom namespace. After creating the provider, be sure to register it in your `app.php` configuration file's `providers` array.

Also, you should register the `Gate` facade in your `app.php` configuration file's `aliases` array:

```
'Gate' => Illuminate\Support\Facades\Gate::class,
```

Update The User Model

Secondly, use the `Illuminate\Foundation\Auth\Access\Authorizable` trait and `Illuminate\Contracts\Auth\Access\Authorizable` contract on your `App\User` model:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Foundation\Auth\Access\Authorizable;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\Access\Authorizable as AuthorizableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements AuthenticatableContract,
                                    AuthorizableContract,
                                    CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword;
}
```

Update The Base Controller

Next, update your base `App\Http\Controllers\Controller` controller to use the `Illuminate\Foundation\Auth\Access\AuthorizesRequests` trait:

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Routing\Controller as BaseController;
use Illuminate\Foundation\Validation\ValidatesRequests;
use Illuminate\Foundation\Auth\Access\AuthorizesRequests;

abstract class Controller extends BaseController
{
    use AuthorizesRequests, DispatchesJobs, ValidatesRequests;
}
```

Upgrading To 5.1.0

Estimated Upgrade Time: Less Than 1 Hour

Update `bootstrap/autoload.php`

Update the `$compiledPath` variable in `bootstrap/autoload.php` to the following:

```
$compiledPath = __DIR__.'/cache/compiled.php';
```

Create `bootstrap/cache` Directory

Within your `bootstrap` directory, create a `cache` directory (`bootstrap/cache`). Place a `.gitignore` file in this directory with the following contents:

```
*
!.gitignore
```

This directory should be writable, and will be used by the framework to store temporary optimization files like `compiled.php`, `routes.php`, `config.php`, and `services.json`.

Add `BroadcastServiceProvider` Provider

Within your `config/app.php` configuration file, add `Illuminate\Broadcasting\BroadcastServiceProvider` to the `providers` array.

Authentication

If you are using the provided `AuthController` which uses the `AuthenticatesAndRegistersUsers` trait, you will need to make a few changes to how new users are validated and created.

First, you no longer need to pass the `Guard` and `Registrar` instances to the base constructor. You can remove these dependencies entirely from your controller's constructor.

Secondly, the `App\Services\Registrar` class used in Laravel 5.0 is no longer needed. You can simply copy and paste your `validator` and `create` method from this class directly into your `AuthController`. No other changes should need to be made to these methods; however, you should be sure to import the `Validator` facade and your `User` model at the top of your `AuthController`.

Password Controller

The included `PasswordController` no longer requires any dependencies in its constructor. You may remove both of the dependencies that were required under 5.0.

Validation

If you are overriding the `formatValidationErrors` method on your base controller class, you should now type-hint the `Illuminate\Contracts\Validation\Validator` contract instead of the concrete `Illuminate\Validation\Validator` instance.

Likewise, if you are overriding the `formatErrors` method on the base form request class, you should now type-hint `Illuminate\Contracts\Validation\Validator` contract instead of the concrete `Illuminate\Validation\Validator` instance.

Eloquent

The `create` Method

Eloquent's `create` method can now be called without any parameters. If you are overriding the `create` method in your own models, set the default value of the `$attributes` parameter to an array:

```
public static function create(array $attributes = [])
{
    // Your custom implementation
}
```

The `find` Method

If you are overriding the `find` method in your own models and calling `parent::find()` within your custom method, you should now change it to call the `find` method on the Eloquent query builder:

```
public static function find($id, $columns = ['*'])
{
    $model = static::query()->find($id, $columns);

    // ...

    return $model;
}
```

The `lists` Method

The `lists` method now returns a `Collection` instance instead of a plain array for Eloquent queries. If you would like to convert the `Collection` into a plain array, use the `all` method:

```
User::lists('id')->all();
```

Be aware that the Query Builder `lists` method still returns an array.

Date Formatting

Previously, the storage format for Eloquent date fields could be modified by overriding the `getDateFormat` method on your model. This is still possible; however, for convenience you may simply specify a `$dateFormat` property on the model instead of overriding the method.

The date format is also now applied when serializing a model to an `array` or JSON. This may change the format of your JSON serialized date fields when migrating from Laravel 5.0 to 5.1. To set a specific date format for serialized models, you may override the `serializeDate(DateTime $date)` method on your model. This method allows you to have granular control over the formatting of serialized Eloquent date fields without changing their storage format.

The Collection Class

The `sort` Method

The `sort` method now returns a fresh collection instance instead of modifying the existing collection:

```
$collection = $collection->sort($callback);
```

The `sortBy` Method

The `sortBy` method now returns a fresh collection instance instead of modifying the existing collection:

```
$collection = $collection->sortBy('name');
```

The `groupBy` Method

The `groupBy` method now returns `Collection` instances for each item in the parent `Collection`. If you would like to convert all of the items back to plain arrays, you may `map` over them:

```
$collection->groupBy('type')->map(function($item)
{
    return $item->all();
});
```

The `lists` Method

The `lists` method now returns a `Collection` instance instead of a plain array. If you would like to convert the `Collection` into a plain array, use the `all` method:

```
$collection->lists('id')->all();
```

Commands & Handlers

The `app/Commands` directory has been renamed to `app/Jobs`. However, you are not required to move all of your commands to the new location, and you may continue using the `make:command` and `handler:command` Artisan commands to generate your classes.

Likewise, the `app/Handlers` directory has been renamed to `app/Listeners` and now only contains event listeners. However, you are not required to move or rename your existing command and event handlers, and you may continue to use the `handler:event` command to generate event handlers.

By providing backwards compatibility for the Laravel 5.0 folder structure, you may upgrade your applications to Laravel 5.1 and slowly upgrade your events and commands to their new locations when it is convenient for you or your team.

Blade

The `createMatcher`, `createOpenMatcher`, and `createPlainMatcher` methods have been removed from the Blade compiler. Use the new `directive` method to create custom directives for Blade in Laravel 5.1. Consult the extending blade documentation for more information.

Tests

Add the protected `$baseUrl` property to the `tests/TestCase.php` file:

```
protected $baseUrl = 'http://localhost';
```

Translation Files

The default directory for published language files for vendor packages has been moved. Move any vendor package language files from `resources/lang/packages/{locale}/{namespace}` to `resources/lang/vendor/{namespace}/{locale}` directory. For example, `Acme/Anvil` package's `acme/anvil::foo` namespaced English language file would be moved from `resources/lang/packages/en/acme/anvil/foo.php` to `resources/lang/vendor/acme/anvil/en/foo.php`.

Amazon Web Services SDK

If you are using the AWS SQS queue driver or the AWS SES e-mail driver, you should update your installed AWS PHP SDK to version 3.0.

If you are using the Amazon S3 filesystem driver, you will need to update the corresponding Flysystem package via Composer:

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`

Deprecations

The following Laravel features have been deprecated and will be removed entirely with the release of Laravel 5.2 in December 2015:

- Route filters have been deprecated in preference of middleware.
- The `Illuminate\Contracts\Routing\Middleware` contract has been deprecated. No contract is required on your middleware. In addition, the `TerminableMiddleware` contract has also been deprecated. Instead of implementing the interface, simply define a `terminate` method on your middleware.
- The `Illuminate\Contracts\Queue\ShouldBeQueued` contract has been deprecated in favor of `Illuminate\Contracts\Queue\ShouldQueue`.
- Iron.io "push queues" have been deprecated in favor of typical Iron.io queues and queue listeners.
- The `Illuminate\Foundation\Bus\DispatchesCommands` trait has been deprecated and renamed to `Illuminate\Foundation\Bus\DispatchesJobs`.
- `Illuminate\Container\BindingResolutionException` has been moved to `Illuminate\Contracts\Container\BindingResolutionException`.
- The service container's `bindShared` method has been deprecated in favor of the `singleton` method.
- The Eloquent and query builder `pluck` method has been deprecated and renamed to `value`.
- The collection `fetch` method has been deprecated in favor of the `pluck` method.
- The `array_fetch` helper has been deprecated in favor of the `array_pluck` method.

Upgrading To 5.0.16

In your `bootstrap/autoload.php` file, update the `$compiledPath` variable to:

```
$compiledPath = __DIR__.'/../vendor/compiled.php';
```

Upgrading To 5.0 From 4.2

Fresh Install, Then Migrate

The recommended method of upgrading is to create a new Laravel `5.0` install and then to copy your `4.2` site's unique application files into the new application. This would include controllers, routes, Eloquent models, Artisan commands, assets, and other code specific files to your application.

To start, install a new Laravel 5.0 application into a fresh directory in your local environment. Do not install any versions newer than 5.0 yet, since we need to complete the migration steps for 5.0 first. We'll discuss each piece of the migration process in further detail below.

Composer Dependencies & Packages

Don't forget to copy any additional Composer dependencies into your 5.0 application. This includes third-party code such as SDKs.

Some Laravel-specific packages may not be compatible with Laravel 5 on initial release. Check with your package's maintainer to determine the proper version of the package for Laravel 5. Once you have added any additional Composer dependencies your application needs, run `composer update`.

Namespacing

By default, Laravel 4 applications did not utilize namespacing within your application code. So, for example, all Eloquent models and controllers simply lived in the "global" namespace. For a quicker migration, you can simply leave these classes in the global namespace in Laravel 5 as well.

Configuration

Migrating Environment Variables

Copy the new `.env.example` file to `.env`, which is the `5.0` equivalent of the old `.env.php` file. Set any appropriate values there, like your `APP_ENV` and `APP_KEY` (your encryption key), your database credentials, and your cache and session drivers.

Additionally, copy any custom values you had in your old `.env.php` file and place them in both `.env` (the real value for your local environment) and `.env.example` (a sample instructional value for other team members).

For more information on environment configuration, view the full documentation.

> **Note:** You will need to place the appropriate `.env` file and values on your production server before deploying your Laravel 5 application.

Configuration Files

Laravel 5.0 no longer uses `app/config/{environmentName}/` directories to provide specific configuration files for a given environment. Instead, move any configuration values that vary by environment into `.env`, and then access them in your configuration files using `env('key', 'default value')`. You will see examples of this in the `config/database.php` configuration file.

Set the config files in the `config/` directory to represent either the values that are consistent across all of your environments, or set them to use `env()` to load values that vary by environment.

Remember, if you add more keys to `.env` file, add sample values to the `.env.example` file as well. This will help your other team members create their own `.env` files.

Routes

Copy and paste your old `routes.php` file into your new `app/Http/routes.php`.

Controllers

Next, move all of your controllers into the `app/Http/Controllers` directory. Since we are not going to migrate to full namespacing in this guide, add the `app/Http/Controllers` directory to the `classmap` directive of your `composer.json` file. Next, you can remove the namespace from the abstract `app/Http/Controllers/Controller.php` base class. Verify that your migrated controllers are extending this base class.

In your `app/Providers/RouteServiceProvider.php` file, set the `namespace` property to `null`.

Route Filters

Copy your filter bindings from `app/filters.php` and place them into the `boot()` method of `app/Providers/RouteServiceProvider.php`. Add `use Illuminate\Support\Facades\Route;` in the `app/Providers/RouteServiceProvider.php` in order to continue using the `Route` Facade.

You do not need to move over any of the default Laravel 4.0 filters such as `auth` and `csrf`; they're all here, but as middleware. Edit any routes or controllers that reference the old default filters (e.g. `['before' => 'auth']`) and change them to reference the new middleware (e.g. `['middleware' => 'auth']`.)

Filters are not removed in Laravel 5. You can still bind and use your own custom filters using `before` and `after`.

Global CSRF

By default, CSRF protection is enabled on all routes. If you'd like to disable this, or only manually enable it on certain routes, remove this line from `App\Http\Kernel`'s `middleware` array:

```
'App\Http\Middleware\VerifyCsrfToken',
```

If you want to use it elsewhere, add this line to `$routeMiddleware`:

```
'csrf' => 'App\Http\Middleware\VerifyCsrfToken',
```

Now you can add the middleware to individual routes / controllers using `['middleware' => 'csrf']` on the route. For more information on middleware, consult the full documentation.

Eloquent Models

Feel free to create a new `app/Models` directory to house your Eloquent models. Again, add this directory to the `classmap` directive of your `composer.json` file.

Update any models using `SoftDeletingTrait` to use `Illuminate\Database\Eloquent\SoftDeletes`.

Eloquent Caching

Eloquent no longer provides the `remember` method for caching queries. You now are responsible for caching your queries manually using the `Cache::remember` function. For more information on caching, consult the full documentation.

User Authentication Model

To upgrade your `User` model for Laravel 5's authentication system, follow these instructions:

Delete the following from your `use` block:

```
use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;
```

Add the following to your `use` block:

```
use Illuminate\Auth\Authenticatable;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
```

Remove the UserInterface and RemindableInterface interfaces.

Mark the class as implementing the following interfaces:

```
implements AuthenticatableContract, CanResetPasswordContract
```

Include the following traits within the class declaration:

```
use Authenticatable, CanResetPassword;
```

If you used them, remove `Illuminate\Auth\Reminders\RemindableTrait` and `Illuminate\Auth\UserTrait` from your use block and your class declaration.

Cashier User Changes

The name of the trait and interface used by Laravel Cashier has changed. Instead of using `BillableTrait`, use the `Laravel\Cashier\Billable` trait. And, instead of `Laravel\Cashier\BillableInterface` implement the `Laravel\Cashier\Contracts\Billable` interface instead. No other method changes are required.

Artisan Commands

Move all of your command classes from your old `app/commands` directory to the new `app/Console/Commands` directory. Next, add the `app/Console/Commands` directory to the `classmap` directive of your `composer.json` file.

Then, copy your list of Artisan commands from `start/artisan.php` into the `command` array of the `app/Console/Kernel.php` file.

Database Migrations & Seeds

Delete the two migrations included with Laravel 5.0, since you should already have the users table in your database.

Move all of your migration classes from the old `app/database/migrations` directory to the new `database/migrations`. All of your seeds should be moved from `app/database/seeds` to `database/seeds`.

Global IoC Bindings

If you have any service container bindings in `start/global.php`, move them all to the `register` method of the `app/Providers/AppServiceProvider.php` file. You may need to import the `App` facade.

Optionally, you may break these bindings up into separate service providers by category.

Views

Move your views from `app/views` to the new `resources/views` directory.

Blade Tag Changes

For better security by default, Laravel 5.0 escapes all output from both the `{{ }}` and `{{{ }}}` Blade directives. A new `{!! !!}` directive has been introduced to display raw, unescaped output. The most secure option when upgrading your application is to only use the new `{!! !!}` directive when you are **certain** that it is safe to display raw output.

However, if you **must** use the old Blade syntax, add the following lines at the bottom of `AppServiceProvider@register`:

```
\Blade::setRawTags('{{', '}}');
\Blade::setContentTags('{{{', '}}}');
\Blade::setEscapedContentTags('{{{', '}}}');
```

This should not be done lightly, and may make your application more vulnerable to XSS exploits. Also, comments with `{{--` will no longer work.

Translation Files

Move your language files from `app/lang` to the new `resources/lang` directory.

Public Directory

Copy your application's public assets from your `4.2` application's `public` directory to your new application's `public` directory. Be sure to keep the `5.0` version of `index.php`.

Tests

Move your tests from `app/tests` to the new `tests` directory.

Misc. Files

Copy in any other files in your project. For example, `.scrutinizer.yml`, `bower.json` and other similar tooling configuration files.

You may move your Sass, Less, or CoffeeScript to any location you wish. The `resources/assets` directory could be a good default location.

Form & HTML Helpers

If you're using Form or HTML helpers, you will see an error stating `class 'Form' not found` or `class 'Html' not found`. The Form and HTML helpers have been deprecated in Laravel 5.0; however, there are community-driven replacements such as those maintained by the Laravel Collective.

For example, you may add `"laravelcollective/html": "~5.0"` to your `composer.json` file's `require` section.

You'll also need to add the Form and HTML facades and service provider. Edit `config/app.php` and add this line to the 'providers' array:

```
'Collective\Html\HtmlServiceProvider',
```

Next, add these lines to the 'aliases' array:

```
'Form' => 'Collective\Html\FormFacade',
'Html' => 'Collective\Html\HtmlFacade',
```

CacheManager

If your application code was injecting `Illuminate\Cache\CacheManager` to get a non-Facade version of Laravel's cache, inject `Illuminate\Contracts\Cache\Repository` instead.

Pagination

Replace any calls to `$paginator->links()` with `$paginator->render()`.

Replace any calls to `$paginator->getFrom()` and `$paginator->getTo()` with `$paginator->firstItem()` and `$paginator->lastItem()` respectively.

Remove the "get" prefix from calls to `$paginator->getPerPage()`, `$paginator->getCurrentPage()`, `$paginator->getLastPage()` and `$paginator->getTotal()` (e.g. `$paginator->perPage()`).

Beanstalk Queuing

Laravel 5.0 now requires `"pda/pheanstalk": "~3.0"` instead of `"pda/pheanstalk": "~2.1"`.

Remote

The Remote component has been deprecated.

Workbench

The Workbench component has been deprecated.

Upgrading To 4.2 From 4.1

PHP 5.4+

Laravel 4.2 requires PHP 5.4.0 or greater.

Encryption Defaults

Add a new `cipher` option in your `app/config/app.php` configuration file. The value of this option should be `MCRYPT_RIJNDAEL_256`.

```
'cipher' => MCRYPT_RIJNDAEL_256
```

This setting may be used to control the default cipher used by the Laravel encryption facilities.

> **Note:** In Laravel 4.2, the default cipher is `MCRYPT_RIJNDAEL_128` (AES), which is considered to be the most secure cipher. Changing the cipher back to `MCRYPT_RIJNDAEL_256` is required to decrypt cookies/values that

were encrypted in Laravel <= 4.1

Soft Deleting Models Now Use Traits

If you are using soft deleting models, the `softDeletes` property has been removed. You must now use the `SoftDeletingTrait` like so:

```
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class User extends Eloquent
{
    use SoftDeletingTrait;
}
```

You must also manually add the `deleted_at` column to your `dates` property:

```
class User extends Eloquent
{
    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];
}
```

The API for all soft delete operations remains the same.

Note: The `SoftDeletingTrait` can not be applied on a base model. It must be used on an actual model class.

View / Pagination Environment Renamed

If you are directly referencing the `Illuminate\View\Environment` class or `Illuminate\Pagination\Environment` class, update your code to reference `Illuminate\View\Factory` and `Illuminate\Pagination\Factory` instead. These two classes have been renamed to better reflect their function.

Additional Parameter On Pagination Presenter

If you are extending the `Illuminate\Pagination\Presenter` class, the abstract method `getPageLinkWrapper` signature has changed to add the `rel` argument:

```
abstract public function getPageLinkWrapper($url, $page, $rel = null);
```

Iron.Io Queue Encryption

If you are using the Iron.io queue driver, you will need to add a new `encrypt` option to your queue configuration file:

```
'encrypt' => true
```

Upgrading To 4.1.29 From <= 4.1.x

Laravel 4.1.29 improves the column quoting for all database drivers. This protects your application from some mass assignment vulnerabilities when **not** using the `fillable` property on models. If you are using the `fillable` property on your models to protect against mass assignment, your application is not vulnerable. However, if you are using `guarded` and are passing a user controlled array into an "update" or "save" type function, you should upgrade to `4.1.29` immediately as your application may be at risk of mass assignment.

To upgrade to Laravel 4.1.29, simply `composer update`. No breaking changes are introduced in this release.

Upgrading To 4.1.26 From <= 4.1.25

Laravel 4.1.26 introduces security improvements for "remember me" cookies. Before this update, if a remember cookie was hijacked by another malicious user, the cookie would remain valid for a long period of time, even after the true owner of the account reset their password, logged out, etc.

This change requires the addition of a new `remember_token` column to your `users` (or equivalent) database table. After this change, a fresh token will be assigned to the user each time they login to your application. The token will also be refreshed when the user logs out of the application. The implications of this change are: if a "remember me" cookie is hijacked, simply logging out of the application will invalidate the cookie.

Upgrade Path

First, add a new, nullable `remember_token` of VARCHAR(100), TEXT, or equivalent to your `users` table.

Next, if you are using the Eloquent authentication driver, update your `User` class with the following three methods:

```
public function getRememberToken()
{
    return $this->remember_token;
}

public function setRememberToken($value)
{
    $this->remember_token = $value;
}

public function getRememberTokenName()
{
    return 'remember_token';
}
```

> **Note:** All existing "remember me" sessions will be invalidated by this change, so all users will be forced to re-authenticate with your application.

Package Maintainers

Two new methods were added to the `Illuminate\Auth\UserProviderInterface` interface. Sample implementations may be found in the default drivers:

```
public function retrieveByToken($identifier, $token);

public function updateRememberToken(UserInterface $user, $token);
```

The `Illuminate\Auth\UserInterface` also received the three new methods described in the "Upgrade Path".

Upgrading To 4.1 From 4.0

Upgrading Your Composer Dependency

To upgrade your application to Laravel 4.1, change your `laravel/framework` version to `4.1.*` in your `composer.json` file.

Replacing Files

Replace your `public/index.php` file with this fresh copy from the repository.

Replace your `artisan` file with this fresh copy from the repository.

Adding Configuration Files & Options

Update your `aliases` and `providers` arrays in your `app/config/app.php` configuration file. The updated values for these arrays can be found in this file. Be sure to add your custom and package service providers / aliases back to the arrays.

Add the new `app/config/remote.php` file from the repository.

Add the new `expire_on_close` configuration option to your `app/config/session.php` file. The default value

should be `false`.

Add the new `failed` configuration section to your `app/config/queue.php` file. Here are the default values for the section:

```
'failed' => [
    'database' => 'mysql', 'table' => 'failed_jobs',
],
```

(Optional) Update the `pagination` configuration option in your `app/config/view.php` file to `pagination::slider-3`.

Controller Updates

If `app/controllers/BaseController.php` has a `use` statement at the top, change `use Illuminate\Routing\Controllers\Controller;` to `use Illuminate\Routing\Controller;`.

Password Reminders Updates

Password reminders have been overhauled for greater flexibility. You may examine the new stub controller by running the `php artisan auth:reminders-controller` Artisan command. You may also browse the updated documentation and update your application accordingly.

Update your `app/lang/en/reminders.php` language file to match this updated file.

Environment Detection Updates

For security reasons, URL domains may no longer be used to detect your application environment. These values are easily spoofable and allow attackers to modify the environment for a request. You should convert your environment detection to use machine host names (`hostname` command on Mac, Linux, and Windows).

Simpler Log Files

Laravel now generates a single log file: `app/storage/logs/laravel.log`. However, you may still configure this behavior in your `app/start/global.php` file.

Removing Redirect Trailing Slash

In your `bootstrap/start.php` file, remove the call to `$app->redirectIfTrailingSlash()`. This method is no longer needed as this functionality is now handled by the `.htaccess` file included with the framework.

Next, replace your Apache `.htaccess` file with this new one that handles trailing slashes.

Current Route Access

The current route is now accessed via `Route::current()` instead of `Route::getCurrentRoute()`.

Composer Update

Once you have completed the changes above, you can run the `composer update` function to update your core application files! If you receive class load errors, try running the `update` command with the `--no-scripts` option enabled like so: `composer update --no-scripts`.

Wildcard Event Listeners

The wildcard event listeners no longer append the event to your handler functions parameters. If you require finding the event that was fired you should use `Event::firing()`.

Prologue

Contribution Guide

Bug Reports

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

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

Remember, bug reports are created in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the bug report will automatically see any activity or that others will jump to fix it. Creating a bug report serves to help yourself and others start on the path of fixing the problem.

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

- Laravel Framework
- Laravel Application
- Laravel Documentation
- Laravel Cashier
- Laravel Envoy
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Website
- Laravel Art

Core Development Discussion

You may propose new features or improvements of existing Laravel behavior in the Laravel Internals issue board. If you propose a new feature, please be willing to implement at least some of the code that would be needed to complete the feature.

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

Which Branch?

All bug fixes should be sent to the latest stable branch or to the current LTS branch (5.1). Bug fixes should **never** be sent to the `master` branch unless they fix features that exist only in the upcoming release.

Minor features that are **fully backwards compatible** with the current Laravel release may be sent to the latest stable branch.

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

If you are unsure if your feature qualifies as a major or minor, please ask Taylor Otwell in the `#internals`

channel of the LaraChat Slack team.

Security Vulnerabilities

If you discover a security vulnerability within Laravel, please send an e-mail 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.

Please note that we do order our imports by length, rather than alphabetically.

PHPDoc

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

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

StyleCI

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

Setup

Installation

Installation

Server Requirements

The Laravel framework has a few system requirements. Of course, all of these requirements are satisfied by the Laravel Homestead virtual machine:

- PHP >= 5.5.9
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

Installing Laravel

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

Via Laravel Installer

First, download the Laravel installer using Composer:

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

Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `laravel` executable can be located by your system.

Once installed, the simple `laravel new` command will create a fresh Laravel installation in the directory you specify. For instance, `laravel new blog` will create a directory named `blog` containing a fresh Laravel installation with all of Laravel's dependencies already installed. This method of installation is much faster than installing via Composer:

```
laravel new blog
```

Via Composer Create-Project

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

```
composer create-project laravel/laravel blog "5.1.*"
```

Configuration

Basic Configuration

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is

documented, so feel free to look through the files and get familiar with the options available to you.

Directory Permissions

After installing Laravel, you may need to configure some permissions. Directories within the `storage` and the `bootstrap/cache` directories should be writable by your web server. If you are using the Homestead virtual machine, these permissions should already be set.

Application Key

The next thing you should do after installing Laravel is set your application key to a random string. If you installed Laravel via Composer or the Laravel installer, this key has already been set for you by the `key:generate` command. Typically, this string should be 32 characters long. The key can be set in the `.env` environment file. If you have not renamed the `.env.example` file to `.env`, you should do that now. **If the application key is not set, your user sessions and other encrypted data will not be secure!**

Additional Configuration

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

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

- Cache
- Database
- Session

Once Laravel is installed, you should also configure your local environment.

Pretty URLs

Apache

The framework ships with a `public/.htaccess` file that is used to allow URLs without `index.php`. If you use Apache to serve your Laravel application, be sure to enable the `mod_rewrite` module.

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

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

On Nginx, the following directive in your site configuration will allow "pretty" URLs:

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

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

Environment Configuration

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

To make this a cinch, Laravel utilizes the DotEnv PHP library by Vance Lucas. In a fresh Laravel installation,

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

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

Feel free to modify your environment variables as needed for your own local server, as well as your production environment. However, your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration.

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

Accessing The Current Application Environment

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

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

You may also pass arguments to the `environment` method to check if the environment matches a given value. You may even pass multiple values if necessary:

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

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

An application instance may also be accessed via the `app` helper method:

```
$environment = app()->environment();
```

Configuration Caching

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

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

Accessing Configuration Values

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

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

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

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

Naming Your Application

After installing Laravel, you may wish to "name" your application. By default, the `app` directory is namespaced under `App`, and autoloaded by Composer using the PSR-4 autoloading standard. However, you may change the

namespace to match the name of your application, which you can easily do via the `app:name` Artisan command.

For example, if your application is named "Horsefly", you could run the following command from the root of your installation:

```
php artisan app:name Horsefly
```

Renaming your application is entirely optional, and you are free to keep the `App` namespace if you wish.

Maintenance Mode

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

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

```
php artisan down
```

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

```
php artisan up
```

Maintenance Mode Response Template

The default template for maintenance mode responses is located in `resources/views/errors/503.blade.php`.

Maintenance Mode & Queues

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

Setup

Laravel Homestead

Introduction

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

Laravel Homestead is an official, pre-packaged Vagrant box that provides you a wonderful development environment without requiring you to install PHP, HHVM, a web server, and any other server software on your local machine. No more worrying about messing up your operating system! Vagrant boxes are completely disposable. If something goes wrong, you can destroy and re-create the box in minutes!

Homestead runs on any Windows, Mac, or Linux system, and includes the Nginx web server, PHP 7.0, MySQL, Postgres, Redis, Memcached, Node, and all of the other goodies you need to develop amazing Laravel applications.

> **Note:** If you are using Windows, you may need to enable hardware virtualization (VT-x). It can usually be enabled via your BIOS. If you are using Hyper-V on a UEFI system you may additionally need to disable Hyper-V in order to access VT-x.

Included Software

- Ubuntu 14.04
- Git
- PHP 7.0
- HHVM
- Nginx
- MySQL
- MariaDB
- Sqlite3
- Postgres
- Composer
- Node (With PM2, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd

Installation & Setup

First Steps

Before launching your Homestead environment, you must install VirtualBox 5.x or VMWare as well as Vagrant. All of these software packages provide easy-to-use visual installers for all popular operating systems.

To use the VMware provider, you will need to purchase both VMware Fusion / Workstation and the VMware Vagrant plug-in. Though it is not free, VMware can provide faster shared folder performance out of the box.

Installing The Homestead Vagrant Box

Once VirtualBox / VMware and Vagrant have been installed, you should add the `laravel/homestead` box to your Vagrant installation using the following command in your terminal. It will take a few minutes to download the box, depending on your Internet connection speed:

```
vagrant box add laravel/homestead
```

If this command fails, make sure your Vagrant installation is up to date.

Installing Homestead

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

```
cd ~

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

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

```
bash init.sh
```

Configuring Homestead

Setting Your Provider

The `provider` key in your `~/.homestead/Homestead.yaml` file indicates which Vagrant provider should be used: `virtualbox`, `vmware_fusion`, or `vmware_workstation`. You may set this to the provider you prefer:

```
provider: virtualbox
```

Configuring Shared Folders

The `folders` property of the `Homestead.yaml` file lists all of the folders you wish to share with your Homestead environment. As files within these folders are changed, they will be kept in sync between your local machine and the Homestead environment. You may configure as many shared folders as necessary:

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

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

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

Configuring Nginx Sites

Not familiar with Nginx? No problem. The `sites` property allows you to easily map a "domain" to a folder on your Homestead environment. A sample site configuration is included in the `Homestead.yaml` file. Again, you may add as many sites to your Homestead environment as necessary. Homestead can serve as a convenient, virtualized environment for every Laravel project you are working on:

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

You can make any Homestead site use HHVM by setting the `hhvm` option to `true`:

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

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

The Hosts File

You must add the "domains" for your Nginx sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead sites into your Homestead machine. On Mac and Linux, this file is located at `/etc/hosts`. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

```
192.168.10.10  homestead.app
```

Make sure the IP address listed is the one set in your `~/.homestead/Homestead.yaml` file. Once you have added the domain to your `hosts` file, you can access the site via your web browser:

```
http://homestead.app
```

Launching The Vagrant Box

Once you have edited the `Homestead.yaml` to your liking, run the `vagrant up` command from your Homestead directory. Vagrant will boot the virtual machine and automatically configure your shared folders and Nginx sites.

To destroy the machine, you may use the `vagrant destroy --force` command.

Per Project Installation

Instead of installing Homestead globally and sharing the same Homestead box across all of your projects, you may instead configure a Homestead instance for each project you manage. Installing Homestead per project may be beneficial if you wish to ship a `Vagrantfile` with your project, allowing others working on the project to simply `vagrant up`.

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

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

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

Mac / Linux:

```
php vendor/bin/homestead make
```

Windows:

vendor\bin\homestead make

Next, run the `vagrant up` command in your terminal and access your project at `http://homestead.app` in your browser. Remember, you will still need to add an `/etc/hosts` file entry for `homestead.app` or the domain of your choice.

Installing MariaDB

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

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

Daily Usage

Accessing Homestead Globally

Sometimes you may want to `vagrant up` your Homestead machine from anywhere on your filesystem. You can do this by adding a simple Bash alias to your Bash profile. This alias will allow you to run any Vagrant command from anywhere on your system and will automatically point that command to your Homestead installation:

```
alias homestead='function __homestead() { (cd ~/Homestead && vagrant $*); unset -f __homestead; };
__homestead'
```

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

Connecting Via SSH

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

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

Connecting To Databases

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

To connect to your MySQL or Postgres database from your host machine via Navicat or Sequel Pro, you should connect to `127.0.0.1` and port `33060` (MySQL) or `54320` (Postgres). The username and password for both databases is `homestead` / `secret`.

> **Note:** You should only use these non-standard ports when connecting to the databases from your host machine. You will use the default 3306 and 5432 ports in your Laravel database configuration file since Laravel is running *within* the virtual machine.

Adding Additional Sites

Once your Homestead environment is provisioned and running, you may want to add additional Nginx sites for your Laravel applications. You can run as many Laravel installations as you wish on a single Homestead environment. To add an additional site, simply add the site to your `~/.homestead/Homestead.yaml` file and then run the `vagrant provision` terminal command from your Homestead directory.

Configuring Cron Schedules

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

If you would like the `schedule:run` command to be run for a Homestead site, you may set the `schedule` option to `true` when defining the site:

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

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

Ports

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

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

Forwarding Additional Ports

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

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

Network Interfaces

The `networks` property of the `Homestead.yaml` configures network interfaces for your Homestead environment. You may configure as many interfaces as necessary:

```
networks:
    - type: "private_network"
      ip: "192.168.10.20"
```

To enable a bridged interface, configure a `bridge` setting and change the network type to `public_network`:

```
networks:
    - type: "public_network"
      ip: "192.168.10.20"
      bridge: "en1: Wi-Fi (AirPort)"
```

To enable DHCP, just remove the `ip` option from your configuration:

```
networks:
    - type: "public_network"
      bridge: "en1: Wi-Fi (AirPort)"
```

Tutorials

Basic Task List

Introduction

This quickstart guide provides a basic introduction to the Laravel framework and includes content on database migrations, the Eloquent ORM, routing, validation, views, and Blade templates. This is a great starting point if you are brand new to the Laravel framework or PHP frameworks in general. If you have already used Laravel or other PHP frameworks, you may wish to consult one of our more advanced quickstarts.

To sample a basic selection of Laravel features, we will build a simple task list we can use to track all of the tasks we want to accomplish (the typical "to-do list" example). The complete, finished source code for this project is available on GitHub.

Installation

Of course, first you will need a fresh installation of the Laravel framework. You may use the Homestead virtual machine or the local PHP environment of your choice to run the framework. Once your local environment is ready, you may install the Laravel framework using Composer:

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

You're free to just read along for the remainder of this quickstart; however, if you would like to download the source code for this quickstart and run it on your local machine, you may clone its Git repository and install its dependencies:

```
git clone https://github.com/laravel/quickstart-basic quickstart
cd quickstart
composer install
php artisan migrate
```

For more complete documentation on building a local Laravel development environment, check out the full Homestead and installation documentation.

Prepping The Database

Database Migrations

First, let's use a migration to define a database table to hold all of our tasks. Laravel's database migrations

provide an easy way to define your database table structure and modifications using fluent, expressive PHP code. Instead of telling your team members to manually add columns to their local copy of the database, your teammates can simply run the migrations you push into source control.

So, let's build a database table that will hold all of our tasks. The Artisan CLI can be used to generate a variety of classes and will save you a lot of typing as you build your Laravel projects. In this case, let's use the `make:migration` command to generate a new database migration for our `tasks` table:

```
php artisan make:migration create_tasks_table --create=tasks
```

The migration will be placed in the `database/migrations` directory of your project. As you may have noticed, the `make:migration` command already added an auto-incrementing ID and timestamps to the migration file. Let's edit this file and add an additional `string` column for the name of our tasks:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTasksTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('tasks');
    }
}
```

To run our migration, we will use the `migrate` Artisan command. If you are using Homestead, you should run this command from within your virtual machine, since your host machine will not have direct access to the database:

```
php artisan migrate
```

This command will create all of our database tables. If you inspect the database tables using the database client of your choice, you should see a new `tasks` table which contains the columns defined in our migration. Next, we're ready to define an Eloquent ORM model for our tasks!

Eloquent Models

Eloquent is Laravel's default ORM (object-relational mapper). Eloquent makes it painless to retrieve and store data in your database using clearly defined "models". Usually, each Eloquent model corresponds directly with a single database table.

So, let's define a `Task` model that corresponds to our `tasks` database table we just created. Again, we can use an Artisan command to generate this model. In this case, we'll use the `make:model` command:

```
php artisan make:model Task
```

The model will be placed in the `app` directory of your application. By default, the model class is empty. We do not have to explicitly tell the Eloquent model which table it corresponds to because it will assume the database table is the plural form of the model name. So, in this case, the `Task` model is assumed to correspond with the

`tasks` database table. Here is what our empty model should look like:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    //
}
```

We'll learn more about how to use Eloquent models as we add routes to our application. Of course, feel free to consult the complete Eloquent documentation for more information.

Routing

Stubbing The Routes

Next, we're ready to add a few routes to our application. Routes are used to point URLs to controllers or anonymous functions that should be executed when a user accesses a given page. By default, all Laravel routes are defined in the `app/Http/routes.php` file that is included in every new project.

For this application, we know we will need at least three routes: a route to display a list of all of our tasks, a route to add new tasks, and a route to delete existing tasks. So, let's stub all of these routes in the `app/Http/routes.php` file:

```
<?php

use App\Task;
use Illuminate\Http\Request;

/**
 * Display All Tasks
 */
Route::get('/', function () {
    //
});

/**
 * Add A New Task
 */
Route::post('/task', function (Request $request) {
    //
});

/**
 * Delete An Existing Task
 */
Route::delete('/task/{id}', function ($id) {
    //
});
```

Displaying A View

Next, let's fill out our `/` route. From this route, we want to render an HTML template that contains a form to add new tasks, as well as a list of all current tasks.

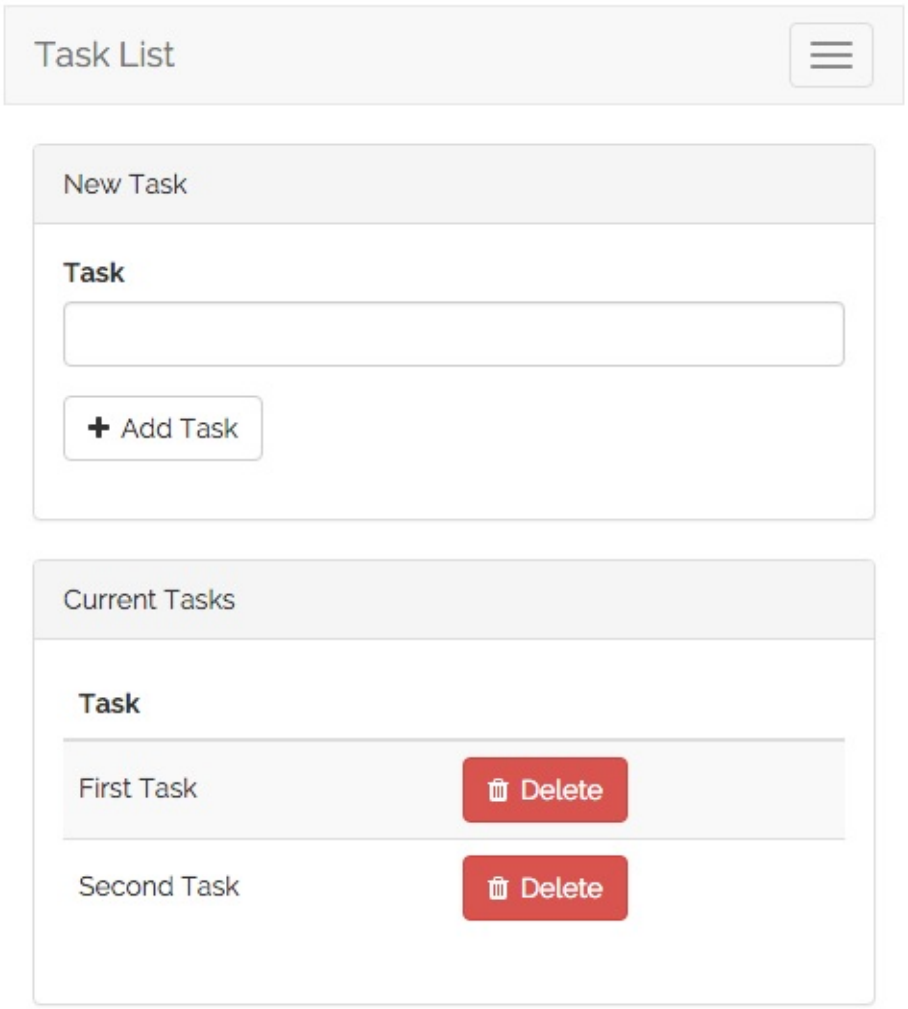
In Laravel, all HTML templates are stored in the `resources/views` directory, and we can use the `view` helper to return one of these templates from our route:

```
Route::get('/', function () {
    return view('tasks');
});
```

Of course, we need to actually define this view, so let's do that now!

Building Layouts & Views

This application only has a single view which contains a form for adding new tasks as well as a listing of all current tasks. To help you visualize the view, here is a screenshot of the finished application with basic Bootstrap CSS styling applied:



Defining The Layout

Almost all web applications share the same layout across pages. For example, this application has a top navigation bar that would be typically present on every page (if we had more than one). Laravel makes it easy to share these common features across every page using Blade **layouts**.

As we discussed earlier, all Laravel views are stored in `resources/views`. So, let's define a new layout view in `resources/views/layouts/app.blade.php`. The `.blade.php` extension instructs the framework to use the Blade templating engine to render the view. Of course, you may use plain PHP templates with Laravel. However, Blade provides convenient short-cuts for writing cleaner, terse templates.

Our `app.blade.php` view should look like the following:

```
// resources/views/layouts/app.blade.php

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Laravel Quickstart - Basic</title>

        <!-- CSS And JavaScript -->
    </head>

    <body>
```

```
        <div class="container">
            <nav class="navbar navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>

        @yield('content')
    </body>
</html>
```

Note the `@yield('content')` portion of the layout. This is a special Blade directive that specifies where all child pages that extend the layout can inject their own content. Next, let's define the child view that will use this layout and provide its primary content.

Defining The Child View

Great, our application layout is finished. Next, we need to define a view that contains a form to create a new task as well as a table that lists all existing tasks. Let's define this view in `resources/views/tasks.blade.php`.

We'll skip over some of the Bootstrap CSS boilerplate and only focus on the things that matter. Remember, you can download the full source for this application on GitHub:

```
// resources/views/tasks.blade.php

@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- Display Validation Errors -->
        @include('common.errors')

        <!-- New Task Form -->
        <form action="/task" method="POST" class="form-horizontal">
            {{ csrf_field() }}

            <!-- Task Name -->
            <div class="form-group">
                <label for="task" class="col-sm-3 control-label">Task</label>

                <div class="col-sm-6">
                    <input type="text" name="name" id="task-name" class="form-control">
                </div>
            </div>

            <!-- Add Task Button -->
            <div class="form-group">
                <div class="col-sm-offset-3 col-sm-6">
                    <button type="submit" class="btn btn-default">
                        <i class="fa fa-plus"></i> Add Task
                    </button>
                </div>
            </div>
        </form>
    </div>

    <!-- TODO: Current Tasks -->
@endsection
```

A Few Notes Of Explanation

Before moving on, let's talk about this template a bit. First, the `@extends` directive informs Blade that we are using the layout we defined at `resources/views/layouts/app.blade.php`. All of the content between `@section('content')` and `@endsection` will be injected into the location of the `@yield('content')` directive within the `app.blade.php` layout.

Now we have defined a basic layout and view for our application. Remember, we are returning this view from our `/` route like so:

```
Route::get('/', function () {
    return view('tasks');
```

```
});
```

Next, we're ready to add code to our `POST /task` route to handle the incoming form input and add a new task to the database.

> **Note:** The `@include('common.errors')` directive will load the template located at `resources/views/common/errors.blade.php`. We haven't defined this template, but we will soon!

Adding Tasks

Validation

Now that we have a form in our view, we need to add code to our `POST /task` route to validate the incoming form input and create a new task. First, let's validate the input.

For this form, we will make the `name` field required and state that it must contain less than `255` characters. If the validation fails, we will redirect the user back to the `/` URL, as well as flash the old input and errors into the session:

```
Route::post('/task', function (Request $request) {
    $validator = Validator::make($request->all(), [
        'name' => 'required|max:255',
    ]);

    if ($validator->fails()) {
        return redirect('/')
            ->withInput()
            ->withErrors($validator);
    }

    // Create The Task...
});
```

The `$errors` Variable

Let's take a break for a moment to talk about the `->withErrors($validator)` portion of this example. The `->withErrors($validator)` call will flash the errors from the given validator instance into the session so that they can be accessed via the `$errors` variable in our view.

Remember that we used the `@include('common.errors')` directive within our view to render the form's validation errors. The `common.errors` will allow us to easily show validation errors in the same format across all of our pages. Let's define the contents of this view now:

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
    <!-- Form Error List -->
    <div class="alert alert-danger">
        <strong>Whoops! Something went wrong!</strong>

        <br><br>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

> **Note:** The `$errors` variable is available in **every** Laravel view. It will simply be an empty instance of `ViewErrorBag` if no validation errors are present.

Creating The Task

Now that input validation is handled, let's actually create a new task by continuing to fill out our route. Once the new task has been created, we will redirect the user back to the `/` URL. To create the task, we may use the

`save` method after creating and setting properties on a new Eloquent model:

```
Route::post('/task', function (Request $request) {
    $validator = Validator::make($request->all(), [
        'name' => 'required|max:255',
    ]);

    if ($validator->fails()) {
        return redirect('/')
            ->withInput()
            ->withErrors($validator);
    }

    $task = new Task;
    $task->name = $request->name;
    $task->save();

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

Great! We can now successfully create tasks. Next, let's continue adding to our view by building a list of all existing tasks.

Displaying Existing Tasks

First, we need to edit our `/` route to pass all of the existing tasks to the view. The `view` function accepts a second argument which is an array of data that will be made available to the view, where each key in the array will become a variable within the view:

```
Route::get('/', function () {
    $tasks = Task::orderBy('created_at', 'asc')->get();

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

Once the data is passed, we can spin through the tasks in our `tasks.blade.php` view and display them in a table. The `@foreach` Blade construct allows us to write concise loops that compile down into blazing fast plain PHP code:

```
@extends('layouts.app')

@section('content')
    <!-- Create Task Form... -->

    <!-- Current Tasks -->
    @if (count($tasks) > 0)
        <div class="panel panel-default">
            <div class="panel-heading">
                Current Tasks
            </div>

            <div class="panel-body">
                <table class="table table-striped task-table">

                    <!-- Table Headings -->
                    <thead>
                        <th>Task</th>
                        <th> </th>
                    </thead>

                    <!-- Table Body -->
                    <tbody>
                        @foreach ($tasks as $task)
                            <tr>
                                <!-- Task Name -->
                                <td class="table-text">
                                    <div>{{ $task->name }}</div>
                                </td>

                                <td>
                                    <!-- TODO: Delete Button -->
                                </td>
                            </tr>
```

```
                    @endforeach
                </tbody>
            </table>
        </div>
    </div>
@endif
@endsection
```

Our task application is almost complete. But, we have no way to delete our existing tasks when they're done. Let's add that next!

Deleting Tasks

Adding The Delete Button

We left a "TODO" note in our code where our delete button is supposed to be. So, let's add a delete button to each row of our task listing within the `tasks.blade.php` view. We'll create a small single-button form for each task in the list. When the button is clicked, a `DELETE /task` request will be sent to the application:

```
<tr>
    <!-- Task Name -->
    <td class="table-text">
        <div>{{ $task->name }}</div>
    </td>

    <!-- Delete Button -->
    <td>
        <form action="/task/{{ $task->id }}" method="POST">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}

            <button>Delete Task</button>
        </form>
    </td>
</tr>
```

A Note On Method Spoofing

Note that the delete button's form `method` is listed as `POST`, even though we are responding to the request using a `Route::delete` route. HTML forms only allow the `GET` and `POST` HTTP verbs, so we need a way to spoof a `DELETE` request from the form.

We can spoof a `DELETE` request by outputting the results of the `method_field('DELETE')` function within our form. This function generates a hidden form input that Laravel recognizes and will use to override the actual HTTP request method. The generated field will look like the following:

```
<input type="hidden" name="_method" value="DELETE">
```

Deleting The Task

Finally, let's add logic to our route to actually delete the given task. We can use the Eloquent `findOrFail` method to retrieve a model by ID or throw a 404 exception if the model does not exist. Once we retrieve the model, we will use the `delete` method to delete the record. Once the record is deleted, we will redirect the user back to the `/` URL:

```
Route::delete('/task/{id}', function ($id) {
    Task::findOrFail($id)->delete();

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

Tutorials

Intermediate Task List

Introduction

This quickstart guide provides an intermediate introduction to the Laravel framework and includes content on database migrations, the Eloquent ORM, routing, authentication, authorization, dependency injection, validation, views, and Blade templates. This is a great starting point if you are familiar with the basics of the Laravel framework or PHP frameworks in general.

To sample a basic selection of Laravel features, we will build a task list we can use to track all of the tasks we want to accomplish (the typical "to-do list" example). In contrast to the "basic" quickstart, this tutorial will allow users to create accounts and authenticate with the application. The complete, finished source code for this project is available on GitHub.

Installation

Of course, first you will need a fresh installation of the Laravel framework. You may use the Homestead virtual machine or the local PHP environment of your choice to run the framework. Once your local environment is ready, you may install the Laravel framework using Composer:

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

You're free to just read along for the remainder of this quickstart; however, if you would like to download the source code for this quickstart and run it on your local machine, you may clone its Git repository and install its dependencies:

```
git clone https://github.com/laravel/quickstart-intermediate quickstart
cd quickstart
composer install
php artisan migrate
```

For more complete documentation on building a local Laravel development environment, check out the full Homestead and installation documentation.

Prepping The Database

Database Migrations

First, let's use a migration to define a database table to hold all of our tasks. Laravel's database migrations provide an easy way to define your database table structure and modifications using fluent, expressive PHP code. Instead of telling your team members to manually add columns to their local copy of the database, your teammates can simply run the migrations you push into source control.

The `users` Table

Since we are going to allow users to create their accounts within the application, we will need a table to store all of our users. Thankfully, Laravel already ships with a migration to create a basic `users` table, so we do not need to manually generate one. The default migration for the `users` table is located in the `database/migrations` directory.

The `tasks` Table

Next, let's build a database table that will hold all of our tasks. The Artisan CLI can be used to generate a variety of classes and will save you a lot of typing as you build your Laravel projects. In this case, let's use the `make:migration` command to generate a new database migration for our `tasks` table:

```
php artisan make:migration create_tasks_table --create=tasks
```

The migration will be placed in the `database/migrations` directory of your project. As you may have noticed, the `make:migration` command already added an auto-incrementing ID and timestamps to the migration file. Let's edit this file and add an additional `string` column for the name of our tasks, as well as a `user_id` column which will link our `tasks` and `users` tables:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTasksTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id')->index();
            $table->string('name');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('tasks');
    }
}
```

To run our migrations, we will use the `migrate` Artisan command. If you are using Homestead, you should run this command from within your virtual machine, since your host machine will not have direct access to the database:

```
php artisan migrate
```

This command will create all of our database tables. If you inspect the database tables using the database client of your choice, you should see new `tasks` and `users` tables which contains the columns defined in our migration. Next, we're ready to define our Eloquent ORM models!

Eloquent Models

Eloquent is Laravel's default ORM (object-relational mapper). Eloquent makes it painless to retrieve and store data in your database using clearly defined "models". Usually, each Eloquent model corresponds directly with a single database table.

The `User` Model

First, we need a model that corresponds to our `users` database table. However, if you look in the `app` directory of your project, you will see that Laravel already ships with a `User` model, so we do not need to generate one manually.

The `Task` Model

So, let's define a `Task` model that corresponds to our `tasks` database table we just created. Again, we can use an Artisan command to generate this model. In this case, we'll use the `make:model` command:

```
php artisan make:model Task
```

The model will be placed in the `app` directory of your application. By default, the model class is empty. We do not have to explicitly tell the Eloquent model which table it corresponds to because it will assume the database table is the plural form of the model name. So, in this case, the `Task` model is assumed to correspond with the `tasks` database table.

Let's add a few things to this model. First, we will state that the `name` attribute on the model should be "mass-assignable":

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

We'll learn more about how to use Eloquent models as we add routes to our application. Of course, feel free to consult the complete Eloquent documentation for more information.

Eloquent Relationships

Now that our models are defined, we need to link them. For example, our `User` can have many `Task` instances, while a `Task` is assigned to one `User`. Defining a relationship will allow us to fluently walk through our relations like so:

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

foreach ($user->tasks as $task) {
    echo $task->name;
}
```

The `tasks` Relationship

First, let's define the `tasks` relationship on our `User` model. Eloquent relationships are defined as methods on

models. Eloquent supports several different types of relationships, so be sure to consult the full Eloquent documentation for more information. In this case, we will define a `tasks` function on the `User` model which calls the `hasMany` method provided by Eloquent:

```
<?php

namespace App;

// Namespace Imports...

class User extends Model implements AuthenticatableContract,
                                    AuthorizableContract,
                                    CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword;

    // Other Eloquent Properties...

    /**
     * Get all of the tasks for the user.
     */
    public function tasks()
    {
        return $this->hasMany(Task::class);
    }
}
```

The `user` Relationship

Next, let's define the `user` relationship on the `Task` model. Again, we will define the relationship as a method on the model. In this case, we will use the `belongsTo` method provided by Eloquent to define the relationship:

```
<?php

namespace App;

use App\User;
use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];

    /**
     * Get the user that owns the task.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Wonderful! Now that our relationships are defined, we can start building our controllers!

Routing

In the basic version of our task list application, we defined all of our logic using Closures within our `routes.php` file. For the majority of this application, we will use controllers to organize our routes. Controllers will allow us to break out HTTP request handling logic across multiple files for better organization.

Displaying A View

We will have a single route that uses a Closure: our `/` route, which will simply be a landing page for application guests. So, let's fill out our `/` route. From this route, we want to render an HTML template that contains the "welcome" page:

In Laravel, all HTML templates are stored in the `resources/views` directory, and we can use the `view` helper to return one of these templates from our route:

```
Route::get('/', function () {
    return view('welcome');
});
```

Of course, we need to actually define this view. We'll do that in a bit!

Authentication

Remember, we also need to let users create accounts and login to our application. Typically, it can be a tedious task to build an entire authentication layer into a web application. However, since it is such a common need, Laravel attempts to make this procedure totally painless.

First, notice that there is already a `app/Http/Controllers/Auth/AuthController` included in your Laravel application. This controller uses a special `AuthenticatesAndRegistersUsers` trait which contains all of the necessary logic to create and authenticate users.

Authentication Routes

So, what's left for us to do? Well, we still need to create the registration and login templates as well as define the routes to point to the authentication controller. First, let's add the routes we need to our `app/Http/routes.php` file:

```
// Authentication Routes...
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Registration Routes...
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
```

Authentication Views

Authentication requires us to create `login.blade.php` and `register.blade.php` within the `resources/views/auth` directory. Of course, the design and styling of these views is unimportant; however, they should at least contain some basic fields.

The `register.blade.php` file should contain a form that includes `name`, `email`, `password`, and `password_confirmation` fields and makes a `POST` request to the `/auth/register` route.

The `login.blade.php` file should contain a form that includes `email` and `password` fields and makes a `POST` request to `/auth/login`.

> **Note:** If you would like to view complete examples for these views, remember that the entire application's source code is available on GitHub.

The Task Controller

Since we know we're going to need to retrieve and store tasks, let's create a `TaskController` using the Artisan CLI, which will place the new controller in the `app/Http/Controllers` directory:

```
php artisan make:controller TaskController --plain
```

Now that the controller has been generated, let's go ahead and stub out some routes in our `app/Http/routes.php` file to point to the controller:

```
Route::get('/tasks', 'TaskController@index');
Route::post('/task', 'TaskController@store');
Route::delete('/task/{task}', 'TaskController@destroy');
```

Authenticating All Task Routes

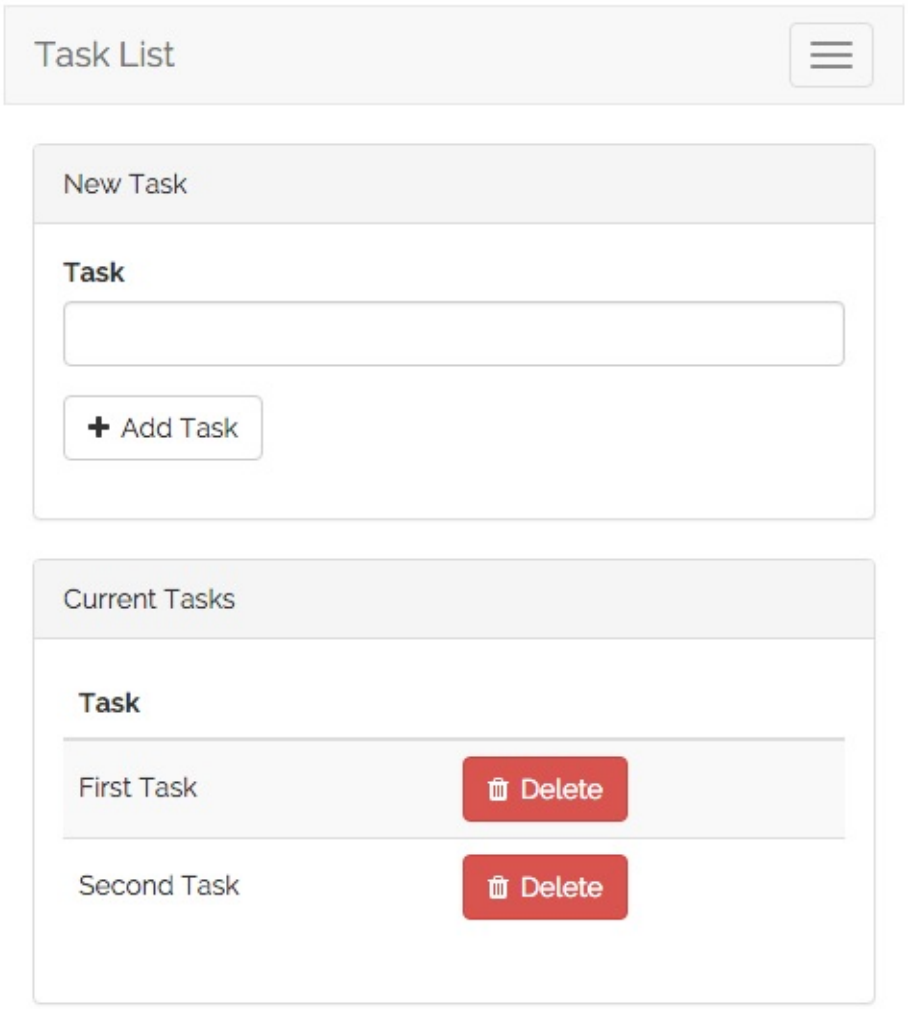
For this application, we want all of our task routes to require an authenticated user. In other words, the user must be "logged into" the application in order to create a task. So, we need to restrict access to our task routes to only authenticated users. Laravel makes this a cinch using middleware.

To require an authenticated users for all actions on the controller, we can add a call to the `middleware` method from the controller's constructor. All available route middleware are defined in the `app/Http/Kernel.php` file. In this case, we want to assign the `auth` middleware to all actions on the controller:

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class TaskController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }
}
```

Building Layouts & Views

This application only has a single view which contains a form for adding new tasks as well as a listing of all current tasks. To help you visualize the view, here is a screenshot of the finished application with basic Bootstrap CSS styling applied:



Defining The Layout

Almost all web applications share the same layout across pages. For example, this application has a top navigation bar that would be typically present on every page (if we had more than one). Laravel makes it easy to share these common features across every page using Blade **layouts**.

As we discussed earlier, all Laravel views are stored in `resources/views`. So, let's define a new layout view in `resources/views/layouts/app.blade.php`. The `.blade.php` extension instructs the framework to use the Blade templating engine to render the view. Of course, you may use plain PHP templates with Laravel. However, Blade provides convenient short-cuts for writing cleaner, terse templates.

Our `app.blade.php` view should look like the following:

```
// resources/views/layouts/app.blade.php

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Laravel Quickstart - Intermediate</title>

        <!-- CSS And JavaScript -->
    </head>

    <body>
        <div class="container">
            <nav class="navbar navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>
```

```
        @yield('content')
    </body>
</html>
```

Note the `@yield('content')` portion of the layout. This is a special Blade directive that specifies where all child pages that extend the layout can inject their own content. Next, let's define the child view that will use this layout and provide its primary content.

Defining The Child View

Great, our application layout is finished. Next, we need to define a view that contains a form to create a new task as well as a table that lists all existing tasks. Let's define this view in `resources/views/tasks/index.blade.php`, which will correspond to the `index` method in our `TaskController`.

We'll skip over some of the Bootstrap CSS boilerplate and only focus on the things that matter. Remember, you can download the full source for this application on GitHub:

```
// resources/views/tasks/index.blade.php

@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- Display Validation Errors -->
        @include('common.errors')

        <!-- New Task Form -->
        <form action="/task" method="POST" class="form-horizontal">
            {{ csrf_field() }}

            <!-- Task Name -->
            <div class="form-group">
                <label for="task-name" class="col-sm-3 control-label">Task</label>

                <div class="col-sm-6">
                    <input type="text" name="name" id="task-name" class="form-control">
                </div>
            </div>

            <!-- Add Task Button -->
            <div class="form-group">
                <div class="col-sm-offset-3 col-sm-6">
                    <button type="submit" class="btn btn-default">
                        <i class="fa fa-plus"></i> Add Task
                    </button>
                </div>
            </div>
        </form>
    </div>

    <!-- TODO: Current Tasks -->
@endsection
```

A Few Notes Of Explanation

Before moving on, let's talk about this template a bit. First, the `@extends` directive informs Blade that we are using the layout we defined at `resources/views/layouts/app.blade.php`. All of the content between `@section('content')` and `@endsection` will be injected into the location of the `@yield('content')` directive within the `app.blade.php` layout.

Now we have defined a basic layout and view for our application. Let's go ahead and return this view from the `index` method of our `TaskController`:

```
/**
 * Display a list of all of the user's task.
 *
 * @param  Request  $request
 * @return Response
```

```
 */
public function index(Request $request)
{
    return view('tasks.index');
}
```

Next, we're ready to add code to our `POST /task` route's controller method to handle the incoming form input and add a new task to the database.

> **Note:** The `@include('common.errors')` directive will load the template located at `resources/views/common/errors.blade.php`. We haven't defined this template, but we will soon!

Adding Tasks

Validation

Now that we have a form in our view, we need to add code to our `TaskController@store` method to validate the incoming form input and create a new task. First, let's validate the input.

For this form, we will make the `name` field required and state that it must contain less than `255` characters. If the validation fails, we want to redirect the user back to the `/tasks` URL, as well as flash the old input and errors into the session:

```
/**
 * Create a new task.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
    ]);

    // Create The Task...
}
```

If you followed along with the basic quickstart, you'll notice this validation code looks quite a bit different! Since we are in a controller, we can leverage the convenience of the `ValidatesRequests` trait that is included in the base Laravel controller. This trait exposes a simple `validate` method which accepts a request and an array of validation rules.

We don't even have to manually determine if the validation failed or do manual redirection. If the validation fails for the given rules, the user will automatically be redirected back to where they came from and the errors will automatically be flashed to the session. Nice!

The `$errors` Variable

Remember that we used the `@include('common.errors')` directive within our view to render the form's validation errors. The `common.errors` will allow us to easily show validation errors in the same format across all of our pages. Let's define the contents of this view now:

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
    <!-- Form Error List -->
    <div class="alert alert-danger">
        <strong>Whoops! Something went wrong!</strong>

        <br><br>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

> **Note:** The `$errors` variable is available in **every** Laravel view. It will simply be an empty instance of `ViewErrorBag` if no validation errors are present.

Creating The Task

Now that input validation is handled, let's actually create a new task by continuing to fill out our route. Once the new task has been created, we will redirect the user back to the `/tasks` URL. To create the task, we are going to leverage the power of Eloquent's relationships.

Most of Laravel's relationships expose a `create` method, which accepts an array of attributes and will automatically set the foreign key value on the related model before storing it in the database. In this case, the `create` method will automatically set the `user_id` property of the given task to the ID of the currently authenticated user, which we are accessing using `$request->user()`:

```
/**
 * Create a new task.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
    ]);

    $request->user()->tasks()->create([
        'name' => $request->name,
    ]);

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

Great! We can now successfully create tasks. Next, let's continue adding to our view by building a list of all existing tasks.

Displaying Existing Tasks

First, we need to edit our `TaskController@index` method to pass all of the existing tasks to the view. The `view` function accepts a second argument which is an array of data that will be made available to the view, where each key in the array will become a variable within the view. For example, we could do this:

```
/**
 * Display a list of all of the user's task.
 *
 * @param  Request  $request
 * @return Response
 */
public function index(Request $request)
{
    $tasks = Task::where('user_id', $request->user()->id)->get();

    return view('tasks.index', [
        'tasks' => $tasks,
    ]);
}
```

However, let's explore some of the dependency injection capabilities of Laravel to inject a `TaskRepository` into our `TaskController`, which we will use for all of our data access.

Dependency Injection

Laravel's service container is one of the most powerful features of the entire framework. After reading this quickstart, be sure to read over all of the container's documentation.

Creating The Repository

As we mentioned earlier, we want to define a `TaskRepository` that holds all of our data access logic for the `Task` model. This will be especially useful if the application grows and you need to share some Eloquent queries across the application.

So, let's create an `app/Repositories` directory and add a `TaskRepository` class. Remember, all Laravel `app` folders are auto-loaded using the PSR-4 auto-loading standard, so you are free to create as many extra directories as needed:

```
<?php

namespace App\Repositories;

use App\User;
use App\Task;

class TaskRepository
{
    /**
     * Get all of the tasks for a given user.
     *
     * @param  User  $user
     * @return Collection
     */
    public function forUser(User $user)
    {
        return Task::where('user_id', $user->id)
                    ->orderBy('created_at', 'asc')
                    ->get();
    }
}
```

Injecting The Repository

Once our repository is defined, we can simply "type-hint" it in the constructor of our `TaskController` and utilize it within our `index` route. Since Laravel uses the container to resolve all controllers, our dependencies will automatically be injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use App\Task;
use App\Http\Requests;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Repositories\TaskRepository;

class TaskController extends Controller
{
    /**
     * The task repository instance.
     *
     * @var TaskRepository
     */
    protected $tasks;

    /**
     * Create a new controller instance.
     *
     * @param  TaskRepository  $tasks
     * @return void
     */
    public function __construct(TaskRepository $tasks)
    {
        $this->middleware('auth');

        $this->tasks = $tasks;
    }

    /**
     * Display a list of all of the user's task.
     *
     * @param  Request  $request
     * @return Response
     */
    public function index(Request $request)
```

```
    {
        return view('tasks.index', [
            'tasks' => $this->tasks->forUser($request->user()),
        ]);
    }
}
```

Displaying The Tasks

Once the data is passed, we can spin through the tasks in our `tasks/index.blade.php` view and display them in a table. The `@foreach` Blade construct allows us to write concise loops that compile down into blazing fast plain PHP code:

```
@extends('layouts.app')

@section('content')
    <!-- Create Task Form... -->

    <!-- Current Tasks -->
    @if (count($tasks) > 0)
        <div class="panel panel-default">
            <div class="panel-heading">
                Current Tasks
            </div>

            <div class="panel-body">
                <table class="table table-striped task-table">

                    <!-- Table Headings -->
                    <thead>
                        <th>Task</th>
                        <th> </th>
                    </thead>

                    <!-- Table Body -->
                    <tbody>
                        @foreach ($tasks as $task)
                            <tr>
                                <!-- Task Name -->
                                <td class="table-text">
                                    <div>{{ $task->name }}</div>
                                </td>

                                <td>
                                    <!-- TODO: Delete Button -->
                                </td>
                            </tr>
                        @endforeach
                    </tbody>
                </table>
            </div>
        </div>
    @endif
@endsection
```

Our task application is almost complete. But, we have no way to delete our existing tasks when they're done. Let's add that next!

Deleting Tasks

Adding The Delete Button

We left a "TODO" note in our code where our delete button is supposed to be. So, let's add a delete button to each row of our task listing within the `tasks/index.blade.php` view. We'll create a small single-button form for each task in the list. When the button is clicked, a `DELETE /task` request will be sent to the application which will trigger our `TaskController@destroy` method:

```
<tr>
    <!-- Task Name -->
    <td class="table-text">
        <div>{{ $task->name }}</div>
    </td>
```

```
    <!-- Delete Button -->
    <td>
        <form action="/task/{{ $task->id }}" method="POST">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}

            <button>Delete Task</button>
        </form>
    </td>
</tr>
```

A Note On Method Spoofing

Note that the delete button's form `method` is listed as `POST`, even though we are responding to the request using a `Route::delete` route. HTML forms only allow the `GET` and `POST` HTTP verbs, so we need a way to spoof a `DELETE` request from the form.

We can spoof a `DELETE` request by outputting the results of the `method_field('DELETE')` function within our form. This function generates a hidden form input that Laravel recognizes and will use to override the actual HTTP request method. The generated field will look like the following:

```
<input type="hidden" name="_method" value="DELETE">
```

Route Model Binding

Now, we're almost ready to define the `destroy` method on our `TaskController`. But, first, let's revisit our route declaration for this route:

```
Route::delete('/task/{task}', 'TaskController@destroy');
```

Without adding any additional code, Laravel would inject the given task ID into the `TaskController@destroy` method, like so:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  string  $taskId
 * @return Response
 */
public function destroy(Request $request, $taskId)
{
    //
}
```

However, the very first thing we will need to do in this method is retrieve the `Task` instance from the database using the given ID. So, wouldn't it be nice if Laravel could just inject the `Task` instance that matches the ID in the first place? Let's make it happen!

In your `app/Providers/RouteServiceProvider.php` file's `boot` method, let's add the following line of code:

```
$router->model('task', 'App\Task');
```

This small line of code will instruct Laravel to retrieve the `Task` model that corresponds to a given ID whenever it sees `{task}` in a route declaration. Now we can define our destroy method like so:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    //
}
```

Authorization

Now, we have a `Task` instance injected into our `destroy` method; however, we have no guarantee that the authenticated user actually "owns" the given task. For example, a malicious request could have been concocted in an attempt to delete another user's tasks by passing a random task ID to the `/tasks/{task}` URL. So, we need to use Laravel's authorization capabilities to make sure the authenticated user actually owns the `Task` instance that was injected into the route.

Creating A Policy

Laravel uses "policies" to organize authorization logic into simple, small classes. Typically, each policy corresponds to a model. So, let's create a `TaskPolicy` using the Artisan CLI, which will place the generated file in `app/Policies/TaskPolicy.php`:

```
php artisan make:policy TaskPolicy
```

Next, let's add a `destroy` method to the policy. This method will receive a `User` instance and a `Task` instance. The method should simply check if the user's ID matches the `user_id` on the task. In fact, all policy methods should either return `true` or `false`:

```
<?php

namespace App\Policies;

use App\User;
use App\Task;
use Illuminate\Auth\Access\HandlesAuthorization;

class TaskPolicy
{
    use HandlesAuthorization;

    /**
     * Determine if the given user can delete the given task.
     *
     * @param  User  $user
     * @param  Task  $task
     * @return bool
     */
    public function destroy(User $user, Task $task)
    {
        return $user->id === $task->user_id;
    }
}
```

Finally, we need to associate our `Task` model with our `TaskPolicy`. We can do this by adding a line in the `app/Providers/AuthServiceProvider.php` file's `$policies` property. This will inform Laravel which policy should be used whenever we try to authorize an action on a `Task` instance:

```
/**
 * The policy mappings for the application.
 *
 * @var array
 */
protected $policies = [
    Task::class => TaskPolicy::class,
];
```

Authorizing The Action

Now that our policy is written, let's use it in our `destroy` method. All Laravel controllers may call an `authorize` method, which is exposed by the `AuthorizesRequest` trait:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    $this->authorize('destroy', $task);
```

```
    // Delete The Task...
}
```

Let's examine this method call for a moment. The first argument passed to the `authorize` method is the name of the policy method we wish to call. The second argument is the model instance that is our current concern. Remember, we recently told Laravel that our `Task` model corresponds to our `TaskPolicy`, so the framework knows on which policy to fire the `destroy` method. The current user will automatically be sent to the policy method, so we do not need to manually pass it here.

If the action is authorized, our code will continue executing normally. However, if the action is not authorized (meaning the policy's `destroy` method returned `false`), a 403 exception will be thrown and an error page will be displayed to the user.

> **Note:** There are several other ways to interact with the authorization services Laravel provides. Be sure to browse the complete authorization documentation.

Deleting The Task

Finally, let's finish adding the logic to our `destroy` method to actually delete the given task. We can use Eloquent's `delete` method to delete the given model instance in the database. Once the record is deleted, we will redirect the user back to the `/tasks` URL:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    $this->authorize('destroy', $task);

    $task->delete();

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

The Basics

HTTP Routing

Basic Routing

You will define most of the routes for your application in the `app/Http/routes.php` file, which is loaded by the `App\Providers\RouteServiceProvider` class. The most basic Laravel routes simply accept a URI and a `Closure`:

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

Route::post('foo/bar', function () {
    return 'Hello World';
});

Route::put('foo/bar', function () {
    //
});

Route::delete('foo/bar', function () {
    //
});
```

Registering A Route For Multiple Verbs

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method on the `Route` facade:

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

Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::any('foo', function () {
    return 'Hello World';
});
```

Generating URLs To Routes

You may generate URLs to your application's routes using the `url` helper:

```
$url = url('foo');
```

Route Parameters

Required Parameters

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

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

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

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

Route parameters are always encased within "curly" braces. The parameters will be passed into your route's `Closure` when the route is executed.

Note: Route parameters cannot contain the `-` character. Use an underscore (_) instead.

Optional Parameters

Occasionally you may need to specify a route parameter, but make the presence of that route parameter optional. You may do so by placing a `?` mark after the parameter name:

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

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

Regular Expression Constraints

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

```
Route::get('user/{name}', function ($name) {
    //
})
->where('name', '[A-Za-z]+');

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

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

Global Constraints

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

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @param  \Illuminate\Routing\Router  $router
 * @return void
 */
public function boot(Router $router)
```

```
{
    $router->pattern('id', '[0-9]+');

    parent::boot($router);
}
```

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

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

Named Routes

Named routes allow you to conveniently generate URLs or redirects for a specific route. You may specify a name for a route using the `as` array key when defining the route:

```
Route::get('user/profile', ['as' => 'profile', function () {
    //
}]);
```

You may also specify route names for controller actions:

```
Route::get('user/profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

Instead of specifying the route name in the route array definition, you may chain the `name` method onto the end of the route definition:

```
Route::get('user/profile', 'UserController@showProfile')->name('profile');
```

Route Groups & Named Routes

If you are using route groups, you may specify an `as` keyword in the route group attribute array, allowing you to set a common route name prefix for all routes within the group:

```
Route::group(['as' => 'admin::'], function () {
    Route::get('dashboard', ['as' => 'dashboard', function () {
        // Route named "admin::dashboard"
    }]);
});
```

Generating URLs To Named Routes

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

```
$url = route('profile');

$redirect = redirect()->route('profile');
```

If the route defines parameters, you may pass the parameters as the second argument to the `route` method. The given parameters will automatically be inserted into the URL:

```
Route::get('user/{id}/profile', ['as' => 'profile', function ($id) {
    //
}]);

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

Route Groups

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

To learn more about route groups, we'll walk through several common use-cases for the feature.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` key in the group attribute array. Middleware will be executed in the order you define this array:

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

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

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers. You may use the `namespace` parameter in your group attribute array to specify the namespace for all controllers within the group:

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

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

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

Sub-Domain Routing

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

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

Route Prefixes

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

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

You may also use the `prefix` parameter to specify common parameters for your grouped routes:

```
Route::group(['prefix' => 'accounts/{account_id}'], function () {
    Route::get('detail', function ($account_id)    {
        // Matches The accounts/{account_id}/detail URL
    });
});
```

CSRF Protection

Introduction

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

Laravel automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application. To generate a hidden input field `_token` containing the CSRF token, you may use the `csrf_field` helper function:

```
<?php echo csrf_field(); ?>
```

The `csrf_field` helper function generates the following HTML:

```
<input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
```

Of course, using the Blade templating engine:

```
{{ csrf_field() }}
```

You do not need to manually verify the CSRF token on POST, PUT, or DELETE requests. The `VerifyCsrfToken` HTTP middleware will verify that the token in the request input matches the token stored in the session.

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 webhook handler route from Laravel's CSRF protection.

You may exclude URIs by adding them to the `$except` property of the `VerifyCsrfToken` middleware:

```
<?php

namespace App\Http\Middleware;

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

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

X-CSRF-TOKEN

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

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

Once you have created the `meta` tag, you can instruct a library like jQuery to add the token to all request headers. This provides simple, convenient CSRF protection for your AJAX based applications:

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

X-XSRF-TOKEN

Laravel also stores the CSRF token in a `XSRF-TOKEN` cookie. You can use the cookie value to set the `X-XSRF-TOKEN` request header. Some JavaScript frameworks, like Angular, do this automatically for you. It is unlikely that you will need to use this value manually.

Route Model Binding

Laravel route model binding provides a convenient way to inject class instances into your routes. For example, instead of injecting a user's ID, you can inject the entire `User` class instance that matches the given ID.

First, use the router's `model` method to specify the class for a given parameter. You should define your model bindings in the `RouteServiceProvider::boot` method:

Binding A Parameter To A Model

```
public function boot(Router $router)
{
    parent::boot($router);

    $router->model('user', 'App\User');
}
```

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

```
$router->get('profile/{user}', function(App\User $user) {
    //
});
```

Since we have bound the `{user}` parameter to the `App\User` model, a `User` instance will be injected into the route. So, for example, a request to `profile/1` will inject the `User` instance which has an ID of 1.

> **Note:** If a matching model instance is not found in the database, a 404 exception will be thrown automatically.

If you wish to specify your own "not found" behavior, pass a Closure as the third argument to the `model` method:

```
$router->model('user', 'App\User', function() {
    throw new NotFoundHttpException;
});
```

If you wish to use your own resolution logic, you should use the `Route::bind` method. The Closure you pass to the `bind` method will receive the value of the URI segment, and should return an instance of the class you want to be injected into the route:

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

Form Method Spoofing

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

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

To generate the hidden input field `_method`, you may also use the `method_field` helper function:

```
<?php echo method_field('PUT'); ?>
```

Of course, using the Blade templating engine:

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

Throwing 404 Errors

There are two ways to manually trigger a 404 error from a route. First, you may use the `abort` helper. The `abort` helper simply throws a `Symfony\Component\HttpFoundation\Exception\HttpException` with the specified status code:

```
abort(404);
```

Secondly, you may manually throw an instance of `Symfony\Component\HttpKernel\Exception\NotFoundHttpException`.

More information on handling 404 exceptions and using custom responses for these errors may be found in the errors section of the documentation.

The Basics

HTTP Middleware

Introduction

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

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

There are several middleware included in the Laravel framework, including middleware for maintenance, authentication, CSRF protection, and more. 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 OldMiddleware
```

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

```
<?php

namespace App\Http\Middleware;

use Closure;

class OldMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') <= 200) {
            return redirect('home');
        }

        return $next($request);
    }

}
```

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

It's best to envision middleware as a series of "layers" HTTP requests must pass through before they hit your

application. Each layer can examine the request and even reject it entirely.

Before / *After* Middleware

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        // Perform action

        return $response;
    }
}
```

Registering Middleware

Global Middleware

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

Assigning Middleware To Routes

If you would like to assign middleware to specific routes, you should first assign the middleware a short-hand key in your `app/Http/Kernel.php` file. By default, the `$routeMiddleware` property of this class contains entries for the middleware included with Laravel. To add your own, simply append it to this list and assign it a key of your choosing. For example:

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

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
];
```

Once the middleware has been defined in the HTTP kernel, you may use the `middleware` key in the route options array:

```
Route::get('admin/profile', ['middleware' => 'auth', function () {
    //
```

```
}]);
```

Use an array to assign multiple middleware to the route:

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

Instead of using an array, you may also chain the `middleware` method onto the route definition:

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

Middleware Parameters

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

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

        return $next($request);
    }

}
```

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

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

Terminable Middleware

Sometimes a middleware may need to do some work after the HTTP response has already been sent to the browser. For example, the "session" middleware included with Laravel writes the session data to storage *after* the response has been sent to the browser. To accomplish this, define the middleware as "terminable" by adding a `terminate` method to the middleware:

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

class StartSession
{
    public function handle($request, Closure $next)
```

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

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

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

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.

The Basics

HTTP Controllers

Introduction

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

Basic Controllers

Here is an example of a basic controller class. All Laravel controllers should extend the base controller class included with the default Laravel installation:

```
<?php

namespace App\Http\Controllers;

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

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

We can route to the controller action like so:

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

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

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. We only defined the portion of the class name that comes after the `App\Http\Controllers` namespace "root". By default, the `RouteServiceProvider` will load the `routes.php` file within a route group containing the root controller namespace.

If you choose to nest or organize your controllers using PHP namespaces deeper into the `App\Http\Controllers` directory, simply use the specific class name relative to the `App\Http\Controllers` root namespace. So, if your

full controller class is `App\Http\Controllers\Photos\AdminController`, you would register a route like so:

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

Naming Controller Routes

Like Closure routes, you may specify names on controller routes:

```
Route::get('foo', ['uses' => 'FooController@method', 'as' => 'name']);
```

URLs To Controller Actions

You may also use the `route` helper to generate a URL to a named controller route:

```
$url = route('name');
```

You may also use the `action` helper method to generate a URL using the controller's class and method names. Again, we only need to specify the part of the controller class name that comes after the base `App\Http\Controllers` namespace:

```
$url = action('FooController@method');
```

You may access the name of the controller action being run using the `currentRouteAction` method on the `Route` facade:

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

Controller Middleware

Middleware may be assigned to the controller's routes like so:

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

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

```
class UserController extends Controller
{
    /**
     * Instantiate a new UserController instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }
}
```

RESTful Resource Controllers

Resource controllers make it painless to build RESTful controllers around resources. For example, you may wish to create a controller that handles HTTP requests regarding "photos" stored by your application. Using the `make:controller` Artisan command, we can quickly create such a controller:

```
php artisan make:controller PhotoController
```

The Artisan command will generate a controller file at `app/Http/Controllers/PhotoController.php`. The controller will contain a method for each of the available resource operations.

Next, you may register a resourceful route to the controller:

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

This single route declaration creates multiple routes to handle a variety of RESTful actions on the photo resource. Likewise, the generated controller will already have methods stubbed for each of these actions, including notes informing you which URIs and verbs they handle.

Actions Handled By Resource Controller

Verb	Path	Action	Route Name
GET	`/photo`	index	photo.index
GET	`/photo/create`	create	photo.create
POST	`/photo`	store	photo.store
GET	`/photo/{photo}`	show	photo.show
GET	`/photo/{photo}/edit`	edit	photo.edit
PUT/PATCH	`/photo/{photo}`	update	photo.update
DELETE	`/photo/{photo}`	destroy	photo.destroy

Partial Resource Routes

When declaring a resource route, you may specify a subset of actions to handle on the route:

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

Route::resource('photo', 'PhotoController',
                ['except' => ['create', 'store', 'update', 'destroy']]);
```

Naming Resource Routes

By default, all resource controller actions have a route name; however, you can override these names by passing a `names` array with your options:

```
Route::resource('photo', 'PhotoController',
                ['names' => ['create' => 'photo.build']]);
```

Nested Resources

Sometimes you may need to define routes to a "nested" resource. For example, a photo resource may have multiple "comments" that may be attached to the photo. To "nest" resource controllers, use "dot" notation in your route declaration:

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

This route will register a "nested" resource that may be accessed with URLs like the following: `photos/{photos}/comments/{comments}`.

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class PhotoCommentController extends Controller
{
    /**
     * Show the specified photo comment.
     *
     * @param  int  $photoId
     * @param  int  $commentId
     * @return Response
     */
    public function show($photoId, $commentId)
    {
        //
```

```
    }
}
```

Supplementing Resource Controllers

If it becomes necessary to add additional routes to a resource controller beyond the default resource routes, you should define those routes before your call to `Route::resource`; otherwise, the routes defined by the `resource` method may unintentionally take precedence over your supplemental routes:

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

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

Implicit Controllers

Laravel allows you to easily define a single route to handle every action in a controller class. First, define the route using the `Route::controller` method. The `controller` method accepts two arguments. The first is the base URI the controller handles, while the second is the class name of the controller:

```
Route::controller('users', 'UserController');
```

Next, just add methods to your controller. The method names should begin with the HTTP verb they respond to followed by the title case version of the URI:

```
<?php

namespace App\Http\Controllers;

class UserController extends Controller
{
    /**
     * Responds to requests to GET /users
     */
    public function getIndex()
    {
        //
    }

    /**
     * Responds to requests to GET /users/show/1
     */
    public function getShow($id)
    {
        //
    }

    /**
     * Responds to requests to GET /users/admin-profile
     */
    public function getAdminProfile()
    {
        //
    }

    /**
     * Responds to requests to POST /users/profile
     */
    public function postProfile()
    {
        //
    }
}
```

As you can see in the example above, `index` methods will respond to the root URI handled by the controller, which, in this case, is `users`.

Assigning Route Names

If you would like to name some of the routes on the controller, you may pass an array of names as the third argument to the `controller` method:

```
Route::controller('users', 'UserController', [
    'getShow' => 'user.show',
]);
```

Dependency Injection & Controllers

Constructor Injection

The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The dependencies will automatically be resolved and injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Repositories\UserRepository;

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

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

Of course, you may also type-hint any Laravel contract. If the container can resolve it, you can type-hint it.

Method Injection

In addition to constructor injection, you may also type-hint dependencies on your controller's action methods. For example, let's type-hint the `Illuminate\Http\Request` instance on one of our methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

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

        //
    }
}
```

If your controller method is also expecting input from a route parameter, simply list your route arguments after your other dependencies. For example, if your route is defined like so:

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

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

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

Route Caching

> **Note:** Route caching does not work with Closure based routes. To use route caching, you must convert any Closure routes to use controller classes.

If your application is exclusively using controller based routes, you may take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. In some cases, your route registration may even be up to 100x faster! To generate a route cache, just execute the `route:cache` Artisan command:

```
php artisan route:cache
```

That's all there is to it! Your cached routes file will now be used instead of your `app/Http/routes.php` file. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you may wish to only run the `route:cache` command during your project's deployment.

To remove the cached routes file without generating a new cache, use the `route:clear` command:

```
php artisan route:clear
```

The Basics

HTTP Requests

Accessing The Request

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

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

        //
    }
}
```

If your controller method is also expecting input from a route parameter, simply list your route arguments after your other dependencies. For example, if your route is defined like so:

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

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

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

Basic Request Information

The `Illuminate\Http\Request` instance provides a variety of methods for examining the HTTP request for your application. The Laravel `Illuminate\Http\Request` extends the `Symfony\Component\HttpFoundation\Request` class. Here are a few more of the useful methods available on this class:

Retrieving The Request URI

The `path` method returns the request's URI. So, if the incoming request is targeted at `http://domain.com/foo/bar`, the `path` method will return `foo/bar`:

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

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

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

To get the full URL, not just the path info, you may use the `url` method on the request instance:

```
$url = $request->url();
```

Retrieving The Request Method

The `method` method will return the HTTP verb for the request. You may also use the `isMethod` method to verify that the HTTP verb matches a given string:

```
$method = $request->method();

if ($request->isMethod('post')) {
    //
}
```

PSR-7 Requests

The PSR-7 standard specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request, you will first need to install a few libraries. Laravel uses the Symfony HTTP Message Bridge component to convert typical Laravel requests and responses into PSR-7 compatible implementations:

```
composer require symfony/psr-http-message-bridge

composer require zendframework/zend-diactoros
```

Once you have installed these libraries, you may obtain a PSR-7 request by simply type-hinting the request type on your route or controller:

```
use Psr\Http\Message\ServerRequestInterface;

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

If you return a PSR-7 response instance from a route or controller, it will automatically be converted back to a Laravel response instance and be displayed by the framework.

Retrieving Input

Retrieving An Input Value

Using a few simple methods, you may access all user input from your `Illuminate\Http\Request` instance. You do not need to worry about the HTTP verb used for the request, as input is accessed in the same way for all

verbs:

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

Alternatively, you may access user input using the properties of the `Illuminate\Http\Request` instance. For example, if one of your application's forms contains a `name` field, you may access the value of the posted field like so:

```
$name = $request->name;
```

You may pass a default value as the second argument to the `input` method. This value will be returned if the requested input value is not present on the request:

```
$name = $request->input('name', 'Sally');
```

When working on forms with array inputs, you may use "dot" notation to access the arrays:

```
$input = $request->input('products.0.name');
```

Determining If An Input Value Is Present

To determine if a value is present on the request, you may use the `has` method. The `has` method returns `true` if the value is present **and** is not an empty string:

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

Retrieving All Input Data

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

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

Retrieving A Portion Of The Input Data

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

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

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

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

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

Old Input

Laravel allows you to keep input from one request during the next request. This feature is particularly useful for re-populating forms after detecting validation errors. However, if you are using Laravel's included validation services, it is unlikely you will need to manually use these methods, as some of Laravel's built-in validation facilities will call them automatically.

Flashing Input To The Session

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

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

You may also use the `flashOnly` and `flashExcept` methods to flash a sub-set of the request data into the session:

```
$request->flashOnly('username', 'email');

$request->flashExcept('password');
```

Flash Input Into Session Then Redirect

Since you often will want to flash input in association with a redirect to the previous page, you may easily chain input flashing onto a redirect using the `withInput` method:

```
return redirect('form')->withInput();

return redirect('form')->withInput($request->except('password'));
```

Retrieving Old Data

To retrieve flashed input from the previous request, use the `old` method on the `Request` instance. The `old` method provides a convenient helper for pulling the flashed input data out of the session:

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

Laravel also provides a global `old` helper function. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper:

```
{{ old('username') }}
```

Cookies

Retrieving Cookies From The Request

All cookies created by the Laravel framework are encrypted and signed with an authentication code, meaning they will be considered invalid if they have been changed by the client. To retrieve a cookie value from the request, you may use the `cookie` method on the `Illuminate\Http\Request` instance:

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

Attaching A New Cookie To A Response

Laravel provides a global `cookie` helper function which serves as a simple factory for generating new `Symfony\Component\HttpFoundation\Cookie` instances. The cookies may be attached to a `Illuminate\Http\Response` instance using the `withCookie` method:

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));

return $response;
```

To create a long-lived cookie, which lasts for five years, you may use the `forever` method on the cookie factory by first calling the `cookie` helper with no arguments, and then chaining the `forever` method onto the returned cookie factory:

```
$response->withCookie(cookie()->forever('name', 'value'));
```

Files

Retrieving Uploaded Files

You may access uploaded files that are included with the `Illuminate\Http\Request` instance using the `file` method. The object returned by the `file` method is an instance of the `Symfony\Component\HttpFoundation\File\UploadedFile` class, which extends the PHP `SplFileInfo` class and provides a variety of methods for interacting with the file:

```
$file = $request->file('photo');
```

Verifying File Presence

You may also determine if a file is present on the request using the `hasFile` method:

```
if ($request->hasFile('photo')) {
    //
}
```

Validating Successful Uploads

In addition to checking if the file is present, you may verify that there were no problems uploading the file via the `isValid` method:

```
if ($request->file('photo')->isValid()) {
    //
}
```

Moving Uploaded Files

To move the uploaded file to a new location, you should use the `move` method. This method will move the file from its temporary upload location (as determined by your PHP configuration) to a more permanent destination of your choosing:

```
$request->file('photo')->move($destinationPath);

$request->file('photo')->move($destinationPath, $fileName);
```

Other File Methods

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

The Basics

HTTP Responses

Basic Responses

Of course, all routes and controllers should return some kind of response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is simply returning a string from a route or controller:

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

The given string will automatically be converted into an HTTP response by the framework.

However, for most routes and controller actions, you will be returning a full `Illuminate\Http\Response` instance or a view. Returning a full `Response` instance allows you to customize the response's HTTP status code and headers. A `Response` instance inherits from the `Symfony\Component\HttpFoundation\Response` class, providing a variety of methods for building HTTP responses:

```
use Illuminate\Http\Response;

Route::get('home', function () {
    return (new Response($content, $status))
                  ->header('Content-Type', $value);
});
```

For convenience, you may also use the `response` helper:

```
Route::get('home', function () {
    return response($content, $status)
                  ->header('Content-Type', $value);
});
```

> **Note:** For a full list of available `Response` methods, check out its API documentation and the Symfony API documentation.

Attaching Headers To Responses

Keep in mind that most response methods are chainable, allowing for the fluent building of responses. For example, you may use the `header` method to add a series of headers to the response before sending it back to the user:

```
return response($content)
            ->header('Content-Type', $type)
            ->header('X-Header-One', 'Header Value')
            ->header('X-Header-Two', 'Header Value');
```

Attaching Cookies To Responses

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

```
return response($content)->header('Content-Type', $type)
                ->withCookie('name', 'value');
```

The `withCookie` method accepts additional optional arguments which allow you to further customize your cookie's properties:

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

By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client. If you would like to disable encryption for a certain subset of cookies generated by your application, you may use the `$except` property of the `App\Http\Middleware\EncryptCookies` middleware:

```
/**
 * The names of the cookies that should not be encrypted.
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

Other Response Types

The `response` helper may be used to conveniently generate other types of response instances. When the `response` helper is called without arguments, an implementation of the `Illuminate\Contracts\Routing\ResponseFactory` contract is returned. This contract provides several helpful methods for generating responses.

View Responses

If you need control over the response status and headers, but also need to return a view as the response content, you may use the `view` method:

```
return response()->view('hello', $data)->header('Content-Type', $type);
```

Of course, if you do not need to pass a custom HTTP status code or custom headers, you may simply use the global `view` helper function.

JSON Responses

The `json` method will automatically set the `Content-Type` header to `application/json`, as well as convert the given array into JSON using the `json_encode` PHP function:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

If you would like to create a JSONP response, you may use the `json` method in addition to `setCallback`:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA'])
                ->setCallback($request->input('callback'));
```

File Downloads

The `download` method may be used to generate a response that forces the user's browser to download the file at the given path. The `download` method accepts a file name as the second argument to the method, which will determine the file name that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

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

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

> **Note:** Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII file name.

Redirects

Redirect responses are instances of the `Illuminate\Http\RedirectResponse` class, and contain the proper headers needed to redirect the user to another URL. There are several ways to generate a `RedirectResponse` instance. The simplest method is to use the global `redirect` helper method:

```
Route::get('dashboard', function () {
    return redirect('home/dashboard');
});
```

Sometimes you may wish to redirect the user to their previous location, for example, after a form submission that is invalid. You may do so by using the global `back` helper function:

```
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', [1]);
```

If you are redirecting to a route with an "ID" parameter that is being populated from an Eloquent model, you may simply pass the model itself. The ID will be extracted automatically:

```
return redirect()->route('profile', [$user]);
```

Redirecting To Controller Actions

You may also generate redirects to controller actions. To do so, simply pass the controller and action name to the `action` method. Remember, you do not need to specify the full namespace to the controller since Laravel's `RouteServiceProvider` will automatically set the default controller namespace:

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

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

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

Redirecting With Flashed Session Data

Redirecting to a new URL and flashing data to the session are typically done at the same time. So, for convenience, you may create a `RedirectResponse` instance **and** flash data to the session in a single method chain. This is particularly convenient for storing status messages after an action:

```
Route::post('user/profile', function () {
    // Update the user's profile...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

Of course, after the user is redirected to a new page, you may retrieve and display the flashed message from the session. For example, using Blade syntax:

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

Response Macros

If you would like to define a custom response that you can re-use in a variety of your routes and controllers, you may use the `macro` method on an implementation of `Illuminate\Contracts\Routing\ResponseFactory`.

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Contracts\Routing\ResponseFactory;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @param  ResponseFactory  $factory
     * @return void
     */
    public function boot(ResponseFactory $factory)
    {
        $factory->macro('caps', function ($value) use ($factory) {
            return $factory->make(strtoupper($value));
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a Closure as its second. The macro's Closure will be executed when calling the macro name from a `ResponseFactory` implementation or the `response` helper:

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

The Basics

Views

Basic Usage

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the `resources/views` directory.

A simple view might look something like this:

```
<!-- View stored in resources/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

Since this view is stored at `resources/views/greeting.php`, we may return it using the global `view` helper function like so:

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

As you can see, the first argument passed to the `view` helper corresponds to the name of the view file in the `resources/views` directory. The second argument passed to helper is an array of data that should be made available to the view. In this case, we are passing the `name` variable, which is displayed in the view by simply executing `echo` on the variable.

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

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

Determining If A View Exists

If you need to determine if a view exists, you may use the `exists` method after calling the `view` helper with no arguments. This method will return `true` if the view exists on disk:

```
if (view()->exists('emails.customer')) {
    //
}
```

When the `view` helper is called without arguments, an instance of `Illuminate\Contracts\View\Factory` is returned, giving you access to any of the factory's methods.

View Data

Passing Data To Views

As you saw in the previous examples, you may easily pass an array of data to views:

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

When passing information in this manner, `$data` should be an array with key/value pairs. Inside your view, you can then access each value using its corresponding key, such as `<?php echo $key; ?>`. As an alternative to

passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view:

```
$view = view('greeting')->with('name', 'Victoria');
```

Sharing Data With All Views

Occasionally, you may need to share a piece of data with all views that are rendered by your application. You may do so using the view factory's `share` method. Typically, you would place calls to `share` within a service provider's `boot` method. You are free to add them to the `AppServiceProvider` or generate a separate service provider to house them:

```
<?php

namespace App\Providers;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        view()->share('key', 'value');
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location.

Let's register our view composers within a service provider. We'll use the `view` helper to access the underlying `Illuminate\Contracts\View\Factory` contract implementation. Remember, Laravel does not include a default directory for view composers. You are free to organize them however you wish. For example, you could create an `App\Http\ViewComposers` directory:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        view()->composer(
            'profile', 'App\Http\ViewComposers\ProfileComposer'
        );

        // Using Closure based composers...
        view()->composer('dashboard', function ($view) {
```

```
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the `providers` array in the `config/app.php` configuration file.

Now that we have registered the composer, the `ProfileComposer@compose` method will be executed each time the `profile` view is being rendered. So, let's define the composer class:

```
<?php

namespace App\Http\ViewComposers;

use Illuminate\Contracts\View\View;
use Illuminate\Users\Repository as UserRepository;

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

    /**
     * Create a new profile composer.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

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

Just before the view is rendered, the composer's `compose` method is called with the `Illuminate\Contracts\View\View` instance. You may use the `with` method to bind data to the view.

> **Note:** All view composers are resolved via the service container, so you may type-hint any dependencies you need within a composer's constructor.

Attaching A Composer To Multiple Views

You may attach a view composer to multiple views at once by passing an array of views as the first argument to the `composer` method:

```
view()->composer(
    ['profile', 'dashboard'],
    'App\Http\ViewComposers\MyViewComposer'
```

```
);
```

The `composer` method accepts the `*` character as a wildcard, allowing you to attach a composer to all views:

```
view()->composer('*', function ($view) {
    //
});
```

View Creators

View **creators** are very similar to view composers; however, they are fired immediately when the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the `creator` method:

```
view()->creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

The Basics

Blade Templates

Introduction

Blade is the simple, yet powerful templating engine provided with Laravel. Unlike other popular PHP templating engines, Blade does not restrict you from using plain PHP code in your views. All Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are *template inheritance* and *sections*. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

Extending A Layout

When defining a child page, you may use the Blade `@extends` directive to specify which layout the child page should "inherit". Views which `@extends` a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

```
<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.master')
```

```
@section('title', 'Page Title')

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

In this example, the `sidebar` section is utilizing the `@@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@@parent` directive will be replaced by the content of the layout when the view is rendered.

Of course, just like plain PHP views, Blade views may be returned from routes using the global `view` helper function:

```
Route::get('blade', function () {
    return view('child');
});
```

Displaying Data

You may display data passed to your Blade views by wrapping the variable in "curly" braces. For example, given the following route:

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

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

Of course, you are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

> **Note:** Blade `{{ }}` statements are automatically sent through PHP's `htmlentities` function to prevent XSS attacks.

Blade & JavaScript Frameworks

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

In this example, the `@` symbol will be removed by Blade; however, `{{ name }}` expression will remain untouched by the Blade engine, allowing it to instead be rendered by your JavaScript framework.

Echoing Data If It Exists

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. We can express this in verbose PHP code like so:

```
{{ isset($name) ? $name : 'Default' }}
```

However, instead of writing a ternary statement, Blade provides you with the following convenient short-cut:

```
{{ $name or 'Default' }}
```

In this example, if the `$name` variable exists, its value will be displayed. However, if it does not exist, the word `Default` will be displayed.

Displaying Unescaped Data

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlentities` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

> **Note:** Be very careful when echoing content that is supplied by users of your application. Always use the double curly brace syntax to escape any HTML entities in the content.

Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient short-cuts for common PHP control structures, such as conditional statements and loops. These short-cuts provide a very clean, terse way of working with PHP control structures, while also remaining familiar to their PHP counterparts.

If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's supported loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

Including Sub-Views

Blade's `@include` directive, allows you to easily include a Blade view from within an existing view. All variables that are available to the parent view will be made available to the included view:

```
<div>
```

```
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

> **Note:** You should avoid using the __DIR__ and __FILE__ constants in your Blade views, since they will refer to the location of the cached view.

Rendering Views For Collections

You may combine loops and includes into one line with Blade's `@each` directive:

```
@each('view.name', $jobs, 'job')
```

The first argument is the view partial to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of `jobs`, typically you will want to access each job as a `job` variable within your view partial.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Service Injection

The `@inject` directive may be used to retrieve a service from the Laravel service container. The first argument passed to `@inject` is the name of the variable the service will be placed into, while the second argument is the class / interface name of the service you wish to resolve:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

Extending Blade

Blade even allows you to define your own custom directives. You can use the `directive` method to register a directive. When the Blade compiler encounters the directive, it calls the provided callback with its parameter.

The following example creates a `@datetime($var)` directive which formats a given `$var`:

```
<?php

namespace App\Providers;

use Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
```

```
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function($expression) {
            return "<?php echo with{$expression}->format('m/d/Y H:i'); ?>";
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

As you can see, Laravel's `with` helper function was used in this directive. The `with` helper simply returns the object / value it is given, allowing for convenient method chaining. The final PHP generated by this directive will be:

```
<?php echo with($var)->format('m/d/Y H:i'); ?>
```

Architecture Foundations

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 Things

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

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

HTTP / Console Kernels

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

The HTTP kernel extends the `Illuminate\Foundation\Http\Kernel` class, which defines an array of `bootstrappers` that will be run before the request is executed. These bootstrappers configure error handling, configure logging, detect the application environment, and perform other tasks that need to be done before the request is actually handled.

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

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

Service Providers

One of the most important Kernel bootstrapping actions is loading the service providers for your application. All of the service providers for the application are configured in the `config/app.php` configuration file's `providers` array. First, the `register` method will be called on all providers, then, once all providers have been registered, the `boot` method will be called.

Service providers are responsible for bootstrapping all of the framework's various components, such as the database, queue, validation, and routing components. Since they bootstrap and configure every feature offered by the framework, service providers are the most important aspect of the entire Laravel bootstrap process.

Dispatch Request

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

Focus On Service Providers

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

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

By default, the `AppServiceProvider` is fairly empty. This provider is a great place to add your application's own bootstrapping and service container bindings. Of course, for large applications, you may wish to create several service providers, each with a more granular type of bootstrapping.

Architecture Foundations

Application Structure

Introduction

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

The Root Directory

The root directory of a fresh Laravel installation contains a variety of folders:

The `app` directory, as you might expect, contains the core code of your application. We'll explore this folder in more detail soon.

The `bootstrap` folder contains a few files that bootstrap the framework and configure autoloading, as well as a `cache` folder that contains a few framework generated files for bootstrap performance optimization.

The `config` directory, as the name implies, contains all of your application's configuration files.

The `database` folder contains your database migration and seeds. If you wish, you may also use this folder to hold an SQLite database.

The `public` directory contains the front controller and your assets (images, JavaScript, CSS, etc.).

The `resources` directory contains your views, raw assets (LESS, SASS, CoffeeScript), and localization files.

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

The `tests` directory contains your automated tests. An example PHPUnit is provided out of the box.

The `vendor` directory contains your Composer dependencies.

The App Directory

The "meat" of your application lives in the `app` directory. By default, this directory is namespaced under `App` and is autoloaded by Composer using the PSR-4 autoloading standard. **You may change this namespace using the `app:name` Artisan command**.

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

The `Jobs` directory, of course, houses the queueable jobs for your application. Jobs may be queued by your application, as well as be run synchronously within the current request lifecycle.

The `Events` directory, as you might expect, houses event classes. Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

The `Listeners` directory contains the handler classes for your events. Handlers receive an event and perform logic in response to the event being fired. For example, a `UserRegistered` event might be handled by a `SendWelcomeEmail` listener.

The `Exceptions` directory contains your application's exception handler and is also a good place to stick any exceptions thrown by your application.

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

Namespacing Your Application

As discussed above, the default application namespace is `App`; however, you may change this namespace to match the name of your application, which is easily done via the `app:name` Artisan command. For example, if your application is named "SocialNet", you would run the following command:

```
php artisan app:name SocialNet
```

Of course, you are free to simply use the `App` namespace.

Architecture Foundations

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. Of course, many of them are "deferred" providers, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

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

Writing Service Providers

All service providers extend the `Illuminate\Support\ServiceProvider` class. This abstract class requires that you define at least one method on your provider: `register`. 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 easily 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 accidently use a service that is provided by a service provider which has not loaded yet.

Now, let's take a look at a basic service provider:

```
<?php

namespace App\Providers;

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

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
```

```
    {
        $this->app->singleton('Riak\Contracts\Connection', function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `Riak\Contracts\Connection` in the service container. If you don't understand how the service container works, check out its documentation.

The Boot Method

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        view()->composer('view', function () {
            //
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Boot Method Dependency Injection

We are able to type-hint dependencies for our `boot` method. The service container will automatically inject any dependencies you need:

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

public function boot(ResponseFactory $factory)
{
    $factory->macro('caps', function ($value) {
        //
    });
}
```

Registering Providers

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

To register your provider, simply add it to the array:

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

    App\Providers\AppServiceProvider::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.

To defer the loading of a provider, set the `defer` property to `true` and define a `provides` method. The `provides` method returns the service container bindings that the provider registers:

```
<?php

namespace App\Providers;

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

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

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

    /**
     * Get the services provided by the provider.
     *
     * @return array
     */
    public function provides()
    {
        return ['Riak\Contracts\Connection'];
    }

}
```

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.

Architecture Foundations

Service Container

Introduction

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

Let's look at a simple example:

```
<?php

namespace App\Jobs;

use App\User;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Contracts\Bus\SelfHandling;

class PurchasePodcast implements SelfHandling
{
    /**
     * The mailer implementation.
     */
    protected $mailer;

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

    /**
     * Purchase a podcast.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

In this example, the `PurchasePodcast` job needs to send e-mails when a podcast is purchased. So, we will **inject** a service that is able to send e-mails. Since the service 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 mailer when testing our application.

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

Binding

Almost all of your service container bindings will be registered within service providers, so all of these

examples will demonstrate using the container in that context. However, 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 such "concrete" objects using PHP's reflection services.

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

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

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

Binding A Singleton

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

```
$this->app->singleton('FooBar', function ($app) {
    return new FooBar($app['SomethingElse']);
});
```

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:

```
$fooBar = new FooBar(new SomethingElse);

$this->app->instance('FooBar', $fooBar);
```

Binding Interfaces To Implementations

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

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

This tells the container that it should inject the `RedisEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in a constructor, or any other location where dependencies are injected by the service container:

```
use App\Contracts\EventPusher;

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

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, when our system receives a new Order, we may want to send an event via PubNub rather than Pusher. Laravel provides a simple, fluent interface for defining this behavior:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
```

```
        ->needs('App\Contracts\EventPusher')
        ->give('App\Services\PubNubEventPusher');
```

You may even pass a Closure to the `give` method:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
          ->needs('App\Contracts\EventPusher')
          ->give(function () {
                  // Resolve dependency...
              });
```

Tagging

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

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

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

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

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

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

Resolving

There are several ways to resolve something out of the container. First, you may use the `make` method, which accepts the name of the class or interface you wish to resolve:

```
$fooBar = $this->app->make('FooBar');
```

Secondly, you may access the container like an array, since it implements PHP's `ArrayAccess` interface:

```
$fooBar = $this->app['FooBar'];
```

Lastly, but most importantly, you may simply "type-hint" the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, queue jobs, middleware, and more. In practice, this is how most of your objects are resolved by the container.

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Users\Repository as UserRepository;

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

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
```

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

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

Container Events

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

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

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

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.

Architecture Foundations

Contracts

Introduction

Laravel's Contracts are a set of interfaces that define the core services provided by the framework. For example, a `Illuminate\Contracts\Queue\Queue` contract defines the methods needed for queueing jobs, while the `Illuminate\Contracts\Mail\Mailer` contract defines the methods needed for sending e-mail.

Each contract has a corresponding implementation provided by the framework. For example, Laravel provides a queue implementation with a variety of drivers, and a mailer implementation that is powered by SwiftMailer.

All of the Laravel contracts live in their own GitHub repository. This provides a quick reference point for all available contracts, as well as a single, decoupled package that may be utilized by package developers.

Contracts Vs. Facades

Laravel's facades provide a simple way of utilizing Laravel's services without needing to type-hint and resolve contracts out of the service container. However, using contracts allows you to define explicit dependencies for your classes. For most applications, using a facade is just fine. However, if you really need the extra loose coupling that contracts can provide, keep reading!

Why Contracts?

You may have several questions regarding contracts. Why use interfaces at all? Isn't using interfaces more complicated? Let's distil the reasons for using interfaces to the following headings: loose coupling and simplicity.

Loose Coupling

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

```
<?php

namespace App\Orders;

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

    /**
     * Create a new repository instance.
     *
     * @param  \SomePackage\Cache\Memcached  $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
        $this->cache = $cache;
    }

    /**
     * Retrieve an Order by ID.
     *
     * @param  int  $id
     * @return Order
```

```
     */
    public function find($id)
    {
        if ($this->cache->has($id))    {
            //
        }
    }
}
```

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

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

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

```
<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

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

    /**
     * Create a new repository instance.
     *
     * @param  Cache  $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}
```

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

Simplicity

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

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

Contract Reference

This is a reference to most Laravel Contracts, as well as their Laravel "facade" counterparts:

Contract	References Facade
Illuminate\Contracts\Auth\Guard	Auth
Illuminate\Contracts\Auth\PasswordBroker	Password
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Broadcasting\Broadcaster	

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

How To Use Contracts

So, how do you get an implementation of a contract? It's actually quite simple.

Many types of classes in Laravel are resolved through the service container, including controllers, event listeners, middleware, queued jobs, and even route Closures. So, to get an implementation of a contract, you can just "type-hint" the interface in the constructor of the class being resolved.

For example, take a look at this event listener:

```
<?php

namespace App\Listeners;

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

class CacheUserInformation
{
    /**
     * The Redis database implementation.
     */
    protected $redis;

    /**
     * Create a new event handler instance.
     *
     * @param  Database  $redis
     * @return void
     */
    public function __construct(Database $redis)
```

```
    {
        $this->redis = $redis;
    }

    /**
     * Handle the event.
     *
     * @param  NewUserRegistered  $event
     * @return void
     */
    public function handle(NewUserRegistered $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.

Architecture Foundations

Facades

Introduction

Facades provide a "static" interface to classes that are available in the application's service container. Laravel ships with many facades, and you have probably been using them without even knowing it! 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.

Using Facades

In the context of 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.

A facade class only needs to implement a single method: `getFacadeAccessor`. It's the `getFacadeAccessor` method's job to define what to resolve from the container. The `Facade` base class makes use of the `__callStatic()` magic-method to defer calls from your facade to the resolved object.

In the example below, a call is made to the Laravel cache system. By glancing at this code, one might assume that the static method `get` is being called on the `Cache` class:

```
<?php

namespace App\Http\Controllers;

use Cache;
use App\Http\Controllers\Controller;

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

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

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

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

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

Instead, the `Cache` facade extends the base `Facade` class and defines the method `getFacadeAccessor()`. Remember, this method's job is to return the name of a service container binding. When a user references any static method on the `Cache` facade, Laravel resolves the `cache` binding from the service container and runs the requested method (in this case, `get`) against that object.

Facade Class Reference

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

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Contracts\Console\Kernel	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	`cache`
Config	Illuminate\Config\Repository	`config`
Cookie	Illuminate\Cookie\CookieJar	`cookie`
Crypt	Illuminate\Encryption\Encrypter	`encrypter`
DB	Illuminate\Database\DatabaseManager	`db`
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	`hash`
Input	Illuminate\Http\Request	`request`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Password	Illuminate\Auth\Passwords\PasswordBroker	`auth.password`
Queue	Illuminate\Queue\QueueManager	`queue`
Queue (Instance)	Illuminate\Queue\QueueInterface	
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\Database	`redis`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Contracts\Routing\ResponseFactory	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	
Storage	Illuminate\Contracts\Filesystem\Factory	`filesystem`
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`
View (Instance)	Illuminate\View\View	

Services

Authentication

Introduction

Laravel makes implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at `config/auth.php`, which contains several well documented options for tweaking the behavior of the authentication services.

Database Considerations

By default, Laravel includes an `App\User` Eloquent model in your `app` directory. This model may be used with the default Eloquent authentication driver. If your application is not using Eloquent, you may use the `database` authentication driver which uses the Laravel query builder.

When building the database schema for the `App\User` model, make sure the password column is at least 60 characters in length.

Also, you should verify that your `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for "remember me" sessions being maintained by your application. This can be done by using `$table->rememberToken();` in a migration.

Authentication Quickstart

Laravel ships with two authentication controllers out of the box, which are located in the `App\Http\Controllers\Auth` namespace. The `AuthController` handles new user registration and authentication, while the `PasswordController` contains the logic to help existing users reset their forgotten passwords. Each of these controllers uses a trait to include their necessary methods. For many applications, you will not need to modify these controllers at all.

Routing

By default, no routes are included to point requests to the authentication controllers. You may manually add them to your `app/Http/routes.php` file:

```
// Authentication routes...
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Registration routes...
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
```

Views

Though the authentication controllers are included with the framework, you will need to provide views that these controllers can render. The views should be placed in the `resources/views/auth` directory. You are free to customize these views however you wish. The login view should be placed at `resources/views/auth/login.blade.php`, and the registration view should be placed at `resources/views/auth/register.blade.php`.

Sample Authentication Form

```
<!-- resources/views/auth/login.blade.php -->

<form method="POST" action="/auth/login">
    {!! csrf_field() !!}

    <div>
        Email
        <input type="email" name="email" value="{{ old('email') }}">
    </div>

    <div>
        Password
        <input type="password" name="password" id="password">
    </div>

    <div>
        <input type="checkbox" name="remember"> Remember Me
    </div>

    <div>
        <button type="submit">Login</button>
    </div>
</form>
```

Sample Registration Form

```
<!-- resources/views/auth/register.blade.php -->

<form method="POST" action="/auth/register">
    {!! csrf_field() !!}

    <div>
        Name
        <input type="text" name="name" value="{{ old('name') }}">
    </div>

    <div>
        Email
        <input type="email" name="email" value="{{ old('email') }}">
    </div>

    <div>
        Password
        <input type="password" name="password">
    </div>

    <div>
        Confirm Password
        <input type="password" name="password_confirmation">
    </div>

    <div>
        <button type="submit">Register</button>
    </div>
</form>
```

Authenticating

Now that you have routes and views setup for the included authentication controllers, you are ready to register and authenticate new users for your application. You may simply access your defined routes in a browser. The authentication controllers already contain the logic (via their traits) to authenticate existing users and store new users in the database.

When a user is successfully authenticated, they will be redirected to the `/home` URI, which you will need to register a route to handle. You can customize the post-authentication redirect location by defining a `redirectPath` property on the `AuthController`:

```
protected $redirectPath = '/dashboard';
```

When a user is not successfully authenticated, they will be redirected to the `/auth/login` URI. You can customize the failed post-authentication redirect location by defining a `loginPath` property on the `AuthController`:

```
protected $loginPath = '/login';
```

The `loginPath` will not change where a user is bounced if they try to access a protected route. That is controlled by the `App\Http\Middleware\Authenticate` middleware's `handle` method.

Customizations

To modify the form fields that are required when a new user registers with your application, or to customize how new user records are inserted into your database, you may modify the `AuthController` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `AuthController` contains the validation rules for new users of the application. You are free to modify this method as you wish.

The `create` method of the `AuthController` is responsible for creating new `App\User` records in your database using the Eloquent ORM. You are free to modify this method according to the needs of your database.

Retrieving The Authenticated User

You may access the authenticated user via the `Auth` facade:

```
$user = Auth::user();
```

Alternatively, once a user is authenticated, you may access the authenticated user via an `Illuminate\Http\Request` instance:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class ProfileController extends Controller
{
    /**
     * Update the user's profile.
     *
     * @param  Request  $request
     * @return Response
     */
    public function updateProfile(Request $request)
    {
        if ($request->user()) {
            // $request->user() returns an instance of the authenticated user...
        }
    }
}
```

Determining If The Current User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method on the `Auth` facade, which will return `true` if the user is authenticated:

```
if (Auth::check()) {
    // The user is logged in...
}
```

However, you may use middleware to verify that the user is authenticated before allowing the user access to certain routes / controllers. To learn more about this, check out the documentation on protecting routes.

Protecting Routes

Route middleware can be used to allow only authenticated users to access a given route. Laravel ships with the `auth` middleware, which is defined in `app\Http\Middleware\Authenticate.php`. All you need to do is attach the middleware to a route definition:

```
// Using A Route Closure...

Route::get('profile', ['middleware' => 'auth', function() {
    // Only authenticated users may enter...
}]);

// Using A Controller...

Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'ProfileController@show'
]);
```

Of course, if you are using controller classes, you may call the `middleware` method from the controller's constructor instead of attaching it in the route definition directly:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Authentication Throttling

If you are using Laravel's built-in `AuthController` class, the `Illuminate\Foundation\Auth\ThrottlesLogins` trait may be used to throttle login attempts to your application. By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts. The throttling is unique to the user's username / e-mail address and their IP address:

```
<?php

namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
    use AuthenticatesAndRegistersUsers, ThrottlesLogins;

    // Rest of AuthController class...
}
```

Manually Authenticating Users

Of course, you are not required to use the authentication controllers included with Laravel. If you choose to remove these controllers, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the `Auth` facade, so we'll need to make sure to import the `Auth` facade at the top of the class. Next, let's check out the `attempt` method:

```
<?php

namespace App\Http\Controllers;

use Auth;
use Illuminate\Routing\Controller;

class AuthController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password])) {
            // Authentication passed...
            return redirect()->intended('dashboard');
        }
    }
}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the hashed `password` value passed to the method via the array. If the two hashed passwords match an authenticated session will be started for the user.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

The `intended` method on the redirector will redirect the user to the URL they were attempting to access before being caught by the authentication filter. A fallback URI may be given to this method in case the intended destination is not available.

If you wish, you also may add extra conditions to the authentication query in addition to the user's e-mail and password. For example, we may verify that user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // The user is active, not suspended, and exists.
}
```

To log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

```
Auth::logout();
```

> **Note:** In these examples, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database.

Remembering Users

If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method, which will keep the user authenticated indefinitely, or until they manually logout. Of course, your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token.

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

If you are "remembering" users, you may use the `viaRemember` method to determine if the user was authenticated using the "remember me" cookie:

```
if (Auth::viaRemember()) {
    //
}
```

Other Authentication Methods

Authenticate A User Instance

If you need to log an existing user instance into your application, you may call the `login` method with the user instance. The given object must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` contract. Of course, the `App\User` model included with Laravel already implements this interface:

```
Auth::login($user);
```

Authenticate A User By ID

To log a user into the application by their ID, you may use the `loginUsingId` method. This method simply accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);
```

Authenticate A User Once

You may use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized, which may be helpful when building a stateless API. The `once` method has the same signature as the `attempt` method:

```
if (Auth::once($credentials)) {
    //
}
```

HTTP Basic Authentication

HTTP Basic Authentication provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the `auth.basic` middleware to your route. The `auth.basic` middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('profile', ['middleware' => 'auth.basic', function() {
    // Only authenticated users may enter...
}]);
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the `auth.basic` middleware will use the `email` column on the user record as the "username".

A Note On FastCGI

If you are using PHP FastCGI, HTTP Basic authentication may not work correctly out of the box. The following lines should be added to your `.htaccess` file:

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session, which is particularly useful for API authentication. To do so, define a middleware that calls the `onceBasic` method. If no response is returned by the `onceBasic` method, the request may be passed further into the application:

```
<?php

namespace Illuminate\Auth\Middleware;

use Auth;
use Closure;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
```

```
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }

}
```

Next, register the route middleware and attach it to a route:

```
Route::get('api/user', ['middleware' => 'auth.basic.once', function() {
    // Only authenticated users may enter...
}]);
```

Resetting Passwords

Database Considerations

Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this on each application, Laravel provides convenient methods for sending password reminders and performing password resets.

To get started, verify that your `App\User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. Of course, the `App\User` model included with the framework already implements this interface, and uses the `Illuminate\Auth\Passwords\CanResetPassword` trait to include the methods needed to implement the interface.

Generating The Reset Token Table Migration

Next, a table must be created to store the password reset tokens. The migration for this table is included with Laravel out of the box, and resides in the `database/migrations` directory. So, all you need to do is migrate:

```
php artisan migrate
```

Routing

Laravel includes an `Auth\PasswordController` that contains the logic necessary to reset user passwords. However, you will need to define routes to point requests to this controller:

```
// Password reset link request routes...
Route::get('password/email', 'Auth\PasswordController@getEmail');
Route::post('password/email', 'Auth\PasswordController@postEmail');

// Password reset routes...
Route::get('password/reset/{token}', 'Auth\PasswordController@getReset');
Route::post('password/reset', 'Auth\PasswordController@postReset');
```

Views

In addition to defining the routes for the `PasswordController`, you will need to provide views that can be returned by this controller. Don't worry, we will provide sample views to help you get started. Of course, you are free to style your forms however you wish.

Sample Password Reset Link Request Form

You will need to provide an HTML view for the password reset request form. This view should be placed at `resources/views/auth/password.blade.php`. This form provides a single field for the user's e-mail address, allowing them to request a password reset link:

```
<!-- resources/views/auth/password.blade.php -->

<form method="POST" action="/password/email">
```

```
    {!! csrf_field() !!}

    @if (count($errors) > 0)
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    @endif

    <div>
        Email
        <input type="email" name="email" value="{{ old('email') }}">
    </div>

    <div>
        <button type="submit">
            Send Password Reset Link
        </button>
    </div>
</form>
```

When a user submits a request to reset their password, they will receive an e-mail with a link that points to the `getReset` method (typically routed at `/password/reset`) of the `PasswordController`. You will need to create a view for this e-mail at `resources/views/emails/password.blade.php`. The view will receive the `$token` variable which contains the password reset token to match the user to the password reset request. Here is an example e-mail view to get you started:

```
<!-- resources/views/emails/password.blade.php -->

Click here to reset your password: {{ url('http://example.com/password/reset/'.$token) }}
```

Sample Password Reset Form

When the user clicks the e-mailed link to reset their password, they will be presented with a password reset form. This view should be placed at `resources/views/auth/reset.blade.php`.

Here is a sample password reset form to get you started:

```
<!-- resources/views/auth/reset.blade.php -->

<form method="POST" action="/password/reset">
    {!! csrf_field() !!}
    <input type="hidden" name="token" value="{{ $token }}">

    @if (count($errors) > 0)
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    @endif

    <div>
        Email
        <input type="email" name="email" value="{{ old('email') }}">
    </div>

    <div>
        Password
        <input type="password" name="password">
    </div>

    <div>
        Confirm Password
        <input type="password" name="password_confirmation">
    </div>

    <div>
        <button type="submit">
            Reset Password
        </button>
    </div>
</form>
```

After Resetting Passwords

Once you have defined the routes and views to reset your user's passwords, you may simply access the routes in your browser. The `PasswordController` included with the framework already includes the logic to send the password reset link e-mails as well as update passwords in the database.

After the password is reset, the user will automatically be logged into the application and redirected to `/home`. You can customize the post password reset redirect location by defining a `redirectTo` property on the `PasswordController`:

```
protected $redirectTo = '/dashboard';
```

> **Note:** By default, password reset tokens expire after one hour. You may change this via the `reminder.expire` option in your `config/auth.php` file.

Social Authentication

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 and Bitbucket.

To get started with Socialite, add to your `composer.json` file as a dependency:

```
composer require laravel/socialite
```

Configuration

After installing the Socialite library, register the `Laravel\Socialite\SocialiteServiceProvider` in your `config/app.php` configuration file:

```
'providers' => [
    // Other service providers...

    Laravel\Socialite\SocialiteServiceProvider::class,
],
```

Also, add the `Socialite` facade to the `aliases` array in your `app` configuration file:

```
'Socialite' => Laravel\Socialite\Facades\Socialite::class,
```

You will also need to add credentials for the OAuth services your application utilizes. These credentials should be placed in your `config/services.php` configuration file, and should use the key `facebook`, `twitter`, `linkedin`, `google`, `github` or `bitbucket`, depending on the providers your application requires. For example:

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

Basic Usage

Next, you are ready to authenticate users! 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. We will access Socialite using the `Socialite` facade:

```
<?php

namespace App\Http\Controllers;

use Socialite;
use Illuminate\Routing\Controller;

class AuthController extends Controller
{
    /**
```

```
     * Redirect the user to the GitHub authentication page.
     *
     * @return Response
     */
    public function redirectToProvider()
    {
        return Socialite::driver('github')->redirect();
    }

    /**
     * Obtain the user information from GitHub.
     *
     * @return Response
     */
    public function handleProviderCallback()
    {
        $user = Socialite::driver('github')->user();

        // $user->token;
    }
}
```

The `redirect` method takes care of sending the user to the OAuth provider, while the `user` method will read the incoming request and retrieve the user's information from the provider. Before redirecting the user, you may also set "scopes" on the request using the `scope` method. This method will overwrite all existing scopes:

```
return Socialite::driver('github')
            ->scopes(['scope1', 'scope2'])->redirect();
```

Of course, you will need to define routes to your controller methods:

```
Route::get('auth/github', 'Auth\AuthController@redirectToProvider');
Route::get('auth/github/callback', 'Auth\AuthController@handleProviderCallback');
```

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:

```
return Socialite::driver('google')
            ->with(['hd' => 'example.com'])->redirect();
```

Retrieving User Details

Once you have a user instance, you can grab a few more details about the user:

```
$user = Socialite::driver('github')->user();

// OAuth Two Providers
$token = $user->token;

// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

Adding Custom Authentication Drivers

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication driver. We will use the `extend` method on the `Auth` facade to define a custom driver. You should place this call to `extend` within a service provider:

```
<?php

namespace App\Providers;

use Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Support\ServiceProvider;
```

```
class AuthServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Auth::extend('riak', function($app) {
            // Return an instance of Illuminate\Contracts\Auth\UserProvider...
            return new RiakUserProvider($app['riak.connection']);
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

After you have registered the driver with the `extend` method, you may switch to the new driver in your `config/auth.php` configuration file.

The User Provider Contract

The `Illuminate\Contracts\Auth\UserProvider` implementations are only responsible for fetching a `Illuminate\Contracts\Auth\Authenticatable` implementation out of a persistent storage system, such as MySQL, Riak, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent it.

Let's take a look at the `Illuminate\Contracts\Auth\UserProvider` contract:

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);

}
```

The `retrieveById` function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The `Authenticatable` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByToken` function retrieves a user by their unique `$identifier` and "remember me" `$token`, stored in a field `remember_token`. As with the previous method, the `Authenticatable` implementation should be returned.

The `updateRememberToken` method updates the `$user` field `remember_token` with the new `$token`. The new token can be either a fresh token, assigned on a successful "remember me" login attempt, or a null when the user is logged out.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to sign into an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition on `$credentials['username']`. The method should then return an implementation of `UserInterface`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given `$user` with the `$credentials` to authenticate the user.

For example, this method might compare the `$user->getAuthPassword()` string to a `Hash::make` of `$credentials['password']`. This method should only validate the user's credentials and return a boolean.

The Authenticatable Contract

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable` contract. Remember, the provider should return implementations of this interface from the `retrieveById` and `retrieveByCredentials` methods:

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable {

    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();

}
```

This interface is simple. The `getAuthIdentifier` method should return the "primary key" of the user. In a MySQL back-end, again, this would be the auto-incrementing primary key. The `getAuthPassword` should return the user's hashed password. This interface allows the authentication system to work with any User class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a `User` class in the `app` directory which implements this interface, so you may consult this class for an implementation example.

Events

Laravel raises a variety of events during the authentication process. You may attach listeners to these events in your `EventServiceProvider`:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    // Fired on each authentication attempt...
    $events->listen('auth.attempt', function ($credentials, $remember, $login) {
        //
    });

    // Fired on successful logins...
    $events->listen('auth.login', function ($user, $remember) {
        //
    });

    // Fired on logouts...
    $events->listen('auth.logout', function ($user) {
        //
    });
}
```

Services

Authorization

Introduction

In addition to providing authentication services out of the box, Laravel also provides a simple way to organize authorization logic and control access to resources. There are a variety of methods and helpers to assist you in organizing your authorization logic, and we'll cover each of them in this document.

> **Note:** Authorization was added in Laravel 5.1.11, please refer to the upgrade guide before integrating these features into your application.

Defining Abilities

The simplest way to determine if a user may perform a given action is to define an "ability" using the `Illuminate\Auth\Access\Gate` class. The `AuthServiceProvider` which ships with Laravel serves as a convenient location to define all of the abilities for your application. For example, let's define an `update-post` ability which receives the current `User` and a `Post` model. Within our ability, we will determine if the user's `id` matches the post's `user_id`:

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);

        $gate->define('update-post', function ($user, $post) {
            return $user->id === $post->user_id;
        });
    }
}
```

Note that we did not check if the given `$user` is not `NULL`. The `Gate` will automatically return `false` for **all abilities** when there is not an authenticated user or a specific user has not been specified using the `forUser` method.

Class Based Abilities

In addition to registering `Closures` as authorization callbacks, you may register class methods by passing a string containing the class name and the method. When needed, the class will be resolved via the service container:

```
$gate->define('update-post', 'Class@method');
```

Intercepting Authorization Checks

Sometimes, you may wish to grant all abilities to a specific user. For this situation, use the `before` method to define a callback that is run before all other authorization checks:

```
$gate->before(function ($user, $ability) {
    if ($user->isSuperAdmin()) {
        return true;
    }
});
```

If the `before` callback returns a non-null result that result will be considered the result of the check.

You may use the `after` method to define a callback to be executed after every authorization check. However, you may not modify the result of the authorization check from an `after` callback:

```
$gate->after(function ($user, $ability, $result, $arguments) {
    //
});
```

Checking Abilities

Via The Gate Facade

Once an ability has been defined, we may "check" it in a variety of ways. First, we may use the `check`, `allows`, or `denies` methods on the `Gate` facade. All of these methods receive the name of the ability and the arguments that should be passed to the ability's callback. You do **not** need to pass the current user to these methods, since the `Gate` will automatically prepend the current user to the arguments passed to the callback. So, when checking the `update-post` ability we defined earlier, we only need to pass a `Post` instance to the `denies` method:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        if (Gate::denies('update-post', $post)) {
            abort(403);
        }

        // Update Post...
    }
}
```

Of course, the `allows` method is simply the inverse of the `denies` method, and returns `true` if the action is authorized. The `check` method is an alias of the `allows` method.

Checking Abilities For Specific Users

If you would like to use the `Gate` facade to check if a user **other than the currently authenticated user** has a given ability, you may use the `forUser` method:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    //
}
```

Passing Multiple Arguments

Of course, ability callbacks may receive multiple arguments:

```
Gate::define('delete-comment', function ($user, $post, $comment) {
    //
});
```

If your ability needs multiple arguments, simply pass an array of arguments to the `Gate` methods:

```
if (Gate::allows('delete-comment', [$post, $comment])) {
    //
}
```

Via The User Model

Alternatively, you may check abilities via the `User` model instance. By default, Laravel's `App\User` model uses an `Authorizable` trait which provides two methods: `can` and `cannot`. These methods may be used similarly to the `allows` and `denies` methods present on the `Gate` facade. So, using our previous example, we may modify our code like so:

```
<?php

namespace App\Http\Controllers;

use App\Post;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        $post = Post::findOrFail($id);

        if ($request->user()->cannot('update-post', $post)) {
            abort(403);
        }

        // Update Post...
    }
}
```

Of course, the `can` method is simply the inverse of the `cannot` method:

```
if ($request->user()->can('update-post', $post)) {
    // Update Post...
}
```

Within Blade Templates

For convenience, Laravel provides the `@can` Blade directive to quickly check if the currently authenticated user has a given ability. For example:

```
<a href="/post/{{ $post->id }}">View Post</a>

@can('update-post', $post)
```

```
    <a href="/post/{{ $post->id }}/edit">Edit Post</a>
@endcan
```

You may also combine the `@can` directive with `@else` directive:

```
@can('update-post', $post)
    <!-- The Current User Can Update The Post -->
@else
    <!-- The Current User Can't Update The Post -->
@endcan
```

Within Form Requests

You may also choose to utilize your `Gate` defined abilities from a form request's `authorize` method. For example:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $postId = $this->route('post');

    return Gate::allows('update', Post::findOrFail($postId));
}
```

Policies

Creating Policies

Since defining all of your authorization logic in the `AuthServiceProvider` could become cumbersome in large applications, Laravel allows you to split your authorization logic into "Policy" classes. Policies are plain PHP classes that group authorization logic based on the resource they authorize.

First, let's generate a policy to manage authorization for our `Post` model. You may generate a policy using the `make:policy` artisan command. The generated policy will be placed in the `app/Policies` directory:

```
php artisan make:policy PostPolicy
```

Registering Policies

Once the policy exists, we need to register it with the `Gate` class. The `AuthServiceProvider` contains a `policies` property which maps various entities to the policies that manage them. So, we will specify that the `Post` model's policy is the `PostPolicy` class:

```
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
```

```
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);
    }
}
```

Writing Policies

Once the policy has been generated and registered, we can add methods for each ability it authorizes. For example, let's define an `update` method on our `PostPolicy`, which will determine if the given `User` can "update" a `Post`:

```
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\Post  $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

You may continue to define additional methods on the policy as needed for the various abilities it authorizes. For example, you might define `show`, `destroy`, or `addComment` methods to authorize various `Post` actions.

> **Note:** All policies are resolved via the Laravel service container, meaning you may type-hint any needed dependencies in the policy's constructor and they will be automatically injected.

Intercepting All Checks

Sometimes, you may wish to grant all abilities to a specific user on a policy. For this situation, define a `before` method on the policy. This method will be run before all other authorization checks on the policy:

```
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

If the `before` method returns a non-null result that result will be considered the result of the check.

Checking Policies

Policy methods are called in exactly the same way as `Closure` based authorization callbacks. You may use the `Gate` facade, the `User` model, the `@can` Blade directive, or the `policy` helper.

Via The Gate Facade

The `Gate` will automatically determine which policy to use by examining the class of the arguments passed to its methods. So, if we pass a `Post` instance to the `denies` method, the `Gate` will utilize the corresponding `PostPolicy` to authorize actions:

```
<?php
```

```
namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        if (Gate::denies('update', $post)) {
            abort(403);
        }

        // Update Post...
    }
}
```

Via The User Model

The `User` model's `can` and `cannot` methods will also automatically utilize policies when they are available for the given arguments. These methods provide a convenient way to authorize actions for any `User` instance retrieved by your application:

```
if ($user->can('update', $post)) {
    //
}

if ($user->cannot('update', $post)) {
    //
}
```

Within Blade Templates

Likewise, the `@can` Blade directive will utilize policies when they are available for the given arguments:

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@endcan
```

Via The Policy Helper

The global `policy` helper function may be used to retrieve the `Policy` class for a given class instance. For example, we may pass a `Post` instance to the `policy` helper to get an instance of our corresponding `PostPolicy` class:

```
if (policy($post)->update($user, $post)) {
    //
}
```

Controller Authorization

By default, the base `App\Http\Controllers\Controller` class included with Laravel uses the `AuthorizesRequests` trait. This trait provides the `authorize` method, which may be used to quickly authorize a given action and throw a `HttpException` if the action is not authorized.

The `authorize` method shares the same signature as the various other authorization methods such as `Gate::allows` and `$user->can()`. So, let's use the `authorize` method to quickly authorize a request to update a `Post`:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Post;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        $this->authorize('update', $post);

        // Update Post...
    }
}
```

If the action is authorized, the controller will continue executing normally; however, if the `authorize` method determines that the action is not authorized, a `HttpException` will automatically be thrown which generates a HTTP response with a `403 Not Authorized` status code. As you can see, the `authorize` method is a convenient, fast way to authorize an action or throw an exception with a single line of code.

The `AuthorizesRequests` trait also provides the `authorizeForUser` method to authorize an action on a user that is not the currently authenticated user:

```
$this->authorizeForUser($user, 'update', $post);
```

Automatically Determining Policy Methods

Frequently, a policy's methods will correspond to the methods on a controller. For example, in the `update` method above, the controller method and the policy method share the same name: `update`.

For this reason, Laravel allows you to simply pass the instance arguments to the `authorize` method, and the ability being authorized will automatically be determined based on the name of the calling function. In this example, since `authorize` is called from the controller's `update` method, the `update` method will also be called on the `PostPolicy`:

```
/**
 * Update the given post.
 *
 * @param  int  $id
 * @return Response
 */
public function update($id)
{
    $post = Post::findOrFail($id);

    $this->authorize($post);

    // Update Post...
}
```

Services

Artisan Console

Introduction

Artisan is the name of the command-line interface included with Laravel. It provides a number of helpful commands for your use while developing your application. It is driven by the powerful Symfony Console component. To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

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

```
php artisan help migrate
```

Writing Commands

In addition to the commands provided with Artisan, you may also build your own custom commands for working with your application. You may store your custom commands in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be autoloaded based on your `composer.json` settings.

To create a new command, you may use the `make:console` Artisan command, which will generate a command stub to help you get started:

```
php artisan make:console SendEmails
```

The command above would generate a class at `app/Console/Commands/SendEmails.php`. When creating the command, the `--command` option may be used to assign the terminal command name:

```
php artisan make:console SendEmails --command=emails:send
```

Command Structure

Once your command is generated, you should fill out the `signature` and `description` properties of the class, which will be used when displaying your command on the `list` screen.

The `handle` method will be called when your command is executed. You may place any command logic in this method. Let's take a look at an example command.

Note that we are able to inject any dependencies we need into the command's constructor. The Laravel service container will automatically inject all dependencies type-hinted in the constructor. For greater code reusability, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks.

```
<?php

namespace App\Console\Commands;
```

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

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

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;

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

        $this->drip = $drip;
    }

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

Command I/O

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.

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

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

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

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

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

Options, like arguments, are also a form of user input. However, they are prefixed by two hyphens (`--`) when they are specified on the command line. We can define options in the signature like so:

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

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

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

You may also specify that the option should be assigned a value by the user by suffixing the option name with a `=` sign, indicating that a value should be provided:

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

In this example, the user may pass a value for the option like so:

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

You may also assign default values to options:

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

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

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

Input Descriptions

You may assign descriptions to input arguments and options by separating the parameter from the description using a colon:

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

Retrieving Input

While your command is executing, you will obviously need to access the values for the arguments and options accepted by your command. To do so, you may use the `argument` and `option` methods:

To retrieve the value of an argument, use the `argument` method:

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

```
    //
}
```

If you need to retrieve all of the arguments as an `array`, call `argument` with no parameters:

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

Options may be retrieved just as easily as arguments using the `option` method. Like the `argument` method, you may call `option` without any parameters in order to retrieve all of the options as an `array`:

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

// Retrieve all options...
$options = $this->option();
```

If the argument or option does not exist, `null` will be returned.

Prompting For Input

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

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

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

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

Asking For Confirmation

If you need to ask the user for a simple confirmation, you may use the `confirm` method. By default, this method will return `false`. However, if the user enters `y` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue? [y|N]')) {
    //
}
```

Giving The User A Choice

The `anticipate` method can be used to provided autocompletion for possible choices. The user can still choose any answer, regardless of the choices.

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

If you need to give the user a predefined set of choices, you may use the `choice` method. The user chooses the index of the answer, but the value of the answer will be returned to you. You may set the default value to be returned if nothing is chosen:

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

Writing Output

To send output to the console, use the `line`, `info`, `comment`, `question` and `error` methods. Each of these methods will use the appropriate ANSI colors for their purpose.

To display an information message to the user, use the `info` method. Typically, this will display in the console as green text:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->info('Display this on the screen');
}
```

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

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

If you want to display plain console output, use the `line` method. The `line` method does not receive any unique coloration:

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

Table Layouts

The `table` method makes it easy to correctly format multiple rows / columns of data. Just pass in the headers and rows to the method. The width and height will be dynamically calculated based on the given data:

```
$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);
```

Progress Bars

For long running tasks, it could be helpful to show a progress indicator. Using the output object, we can start, advance and stop the Progress Bar. You have to define the number of steps when you start the progress, then advance the Progress Bar after each step:

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

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

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

    $bar->advance();
}

$bar->finish();
```

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

Registering Commands

Once your command is finished, you need to register it with Artisan so it will be available for use. This is done within the `app/Console/Kernel.php` file.

Within this file, you will find a list of commands in the `commands` property. To register your command, simply add the class name to the list. When Artisan boots, all the commands listed in this property will be resolved by the service container and registered with Artisan:

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

Calling Commands Via Code

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

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

Using the `queue` method on the `Artisan` facade, you may even queue Artisan commands so they are processed in the background by your queue workers:

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

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

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

Calling Commands From Other Commands

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

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

    //
}
```

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

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

Services

Laravel Cashier

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.

Configuration

Composer

First, add the Cashier package to your `composer.json` file and run the `composer update` command:

```
"laravel/cashier": "~5.0" (For Stripe SDK ~2.0, and Stripe APIs on 2015-02-18 version and later)
"laravel/cashier": "~4.0" (For Stripe APIs on 2015-02-18 version and later)
"laravel/cashier": "~3.0" (For Stripe APIs up to and including 2015-02-16 version)
```

Service Provider

Next, register the `Laravel\Cashier\CashierServiceProvider` service provider in your `app` configuration file.

Migration

Before using Cashier, we'll need to add several columns to your database. Don't worry, you can use the `cashier:table` Artisan command to create a migration to add the necessary column. For example, to add the column to the users table run the command: `php artisan cashier:table users`.

Once the migration has been created, simply run the `migrate` command.

Model Setup

Next, add the `Billable` trait and appropriate date mutators to your model definition:

```
use Laravel\Cashier\Billable;
use Laravel\Cashier\Contracts\Billable as BillableContract;

class User extends Model implements BillableContract
{
    use Billable;

    protected $dates = ['trial_ends_at', 'subscription_ends_at'];
```

```
}
```

Adding the columns to your model's `$dates` property will instruct Eloquent to return the columns as Carbon / DateTime instances instead of raw strings.

Stripe Key

Finally, set your Stripe key in your `services.php` configuration file:

```
'stripe' => [
    'model'  => 'User',
    'secret' => env('STRIPE_API_SECRET'),
],
```

Subscriptions

Creating Subscriptions

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

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

$user->subscription('monthly')->create($creditCardToken);
```

The `create` method will automatically create the Stripe subscription, as well as update your database with Stripe customer ID and other relevant billing information. If your plan has a trial configured in Stripe, the trial end date will also automatically be set on the user record.

If you want to implement trial periods, but are managing the trials entirely within your application instead of defining them within Stripe, you must manually set the trial end date:

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

Additional User Details

If you would like to specify additional customer details, you may do so by passing them as the second argument to the `create` method:

```
$user->subscription('monthly')->create($creditCardToken, [
    'email' => $email, 'description' => 'Our First Customer'
]);
```

To learn more about the additional fields supported by Stripe, check out Stripe's documentation on customer creation.

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$user->subscription('monthly')
     ->withCoupon('code')
     ->create($creditCardToken);
```

Checking Subscription Status

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

```
if ($user->subscribed()) {
```

```
    //
}
```

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

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed()) {
        // This user is not a paying customer...
        return redirect('billing');
    }

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

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

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

The `onPlan` method may be used to determine if the user is subscribed to a given plan based on its Stripe ID:

```
if ($user->onPlan('monthly')) {
    //
}
```

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->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->onGracePeriod()) {
    //
}
```

The `everSubscribed` method may be used to determine if the user has ever subscribed to a plan in your application:

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

Changing Plans

After a user is subscribed to your application, they may occasionally want to change to a new subscription plan. To swap a user to a new subscription, use the `swap` method. For example, we may easily switch a user to the `premium` subscription:

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

$user->subscription('premium')->swap();
```

If the user is on trial, the trial period will be maintained. Also, if a "quantity" exists for the subscription, that quantity will also be maintained. When swapping plans, you may also use the `prorate` method to indicate that the charges should be pro-rated. In addition, you may use the `swapAndInvoice` method to immediately invoice the user for the plan change:

```
$user->subscription('premium')
            ->prorate()
            ->swapAndInvoice();
```

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, your application might charge $10 per month **per user** on an account. To easily increment or decrement your subscription quantity, use the `increment` and `decrement` methods:

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

$user->subscription()->increment();

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

$user->subscription()->decrement();

// Subtract five to the subscription's current quantity...
$user->subscription()->decrement(5);
```

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

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

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

Subscription Taxes

With Cashier, it's easy to provide the `tax_percent` value sent to Stripe. To specify the tax percentage a user pays on a subscription, implement the `getTaxPercent` method on your billable model, and return a numeric value between 0 and 100, with no more than 2 decimal places.

```
public function getTaxPercent() {
    return 20;
}
```

This enables you to apply a tax rate on a model-by-model basis, which may be helpful for a user base that spans multiple countries.

Cancelling Subscriptions

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

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

When a subscription is cancelled, Cashier will automatically set the `subscription_ends_at` column in your database. This column is used to know when the `subscribed` method should begin returning `false`. For example, if a customer cancels a subscription on March 1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th.

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

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

Resuming Subscriptions

If a user has cancelled their subscription and you wish to resume it, use the `resume` method:

```
$user->subscription('monthly')->resume($creditCardToken);
```

If the user cancels a subscription and then resumes that subscription before the subscription has fully expired,

they will not be billed immediately. Instead, their subscription will simply be re-activated, and they will be billed on the original billing cycle.

Handling Stripe Webhooks

Failed Subscriptions

What if a customer's credit card expires? No worries - Cashier includes a Webhook controller that can easily cancel the customer's subscription for you. Just point a route to the controller:

```
Route::post('stripe/webhook', '\Laravel\Cashier\WebhookController@handleWebhook');
```

That's it! Failed payments will be captured and handled by the controller. The controller will cancel the customer's subscription when Stripe determines the subscription has failed (normally after three failed payment attempts). Don't forget: you will need to configure the webhook URI in your Stripe control panel settings.

Since Stripe webhooks need to bypass Laravel's CSRF verification, be sure to list the URI as an exception in your `VerifyCsrfToken` middleware:

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

Other Webhooks

If you have additional Stripe webhook events you would like to handle, simply extend the Webhook controller. Your method names should correspond to Cashier's expected convention, specifically, methods should be prefixed with `handle` and the "camel case" name of the Stripe webhook you wish to handle. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you should add a `handleInvoicePaymentSucceeded` method to the controller.

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\WebhookController as BaseController;

class WebhookController extends BaseController
{
    /**
     * Handle a stripe webhook.
     *
     * @param  array  $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

Single Charges

If you would like to make a "one off" charge against a subscribed customer's credit card, you may use the `charge` method on a billable model instance. The `charge` method accepts the amount you would like to charge in the **lowest denominator of the currency used by your application**. So, for example, the example below will charge 100 cents, or $1.00, against the user's credit card:

```
$user->charge(100);
```

The `charge` method accepts an array as its second argument, allowing you to pass any options you wish to the underlying Stripe charge creation:

```
$user->charge(100, [
    'source' => $token,
    'receipt_email' => $user->email,
]);
```

The `charge` method will return `false` if the charge fails. This typically indicates the charge was denied:

```
if ( ! $user->charge(100)) {
    // The charge was denied...
}
```

If the charge is successful, the full Stripe response will be returned from the method.

Invoices

You may easily retrieve an array of a billable model's invoices using the `invoices` method:

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

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

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

Generating Invoice PDFs

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

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

Services

Cache

Configuration

Laravel provides a unified API for various caching systems. The cache configuration is located at `config/cache.php`. In this file you may specify which cache driver you would like used by default throughout your application. Laravel supports popular caching backends like Memcached and Redis out of the box.

The cache configuration file also contains various other options, which are documented within the file, so make sure to read over these options. By default, Laravel is configured to use the `file` cache driver, which stores the serialized, cached objects in the filesystem. For larger applications, it is recommended that you use an in-memory cache such as Memcached or APC. You may even configure multiple cache configurations for the same driver.

Cache Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain the cache items. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache', function($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

Memcached

Using the Memcached cache requires the Memcached PECL package to be installed.

The default configuration uses TCP/IP based on Memcached::addServer:

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

You may also set the `host` option to a UNIX socket path. If you do this, the `port` option should be set to `0`:

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

Redis

Before using a Redis cache with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer.

For more information on configuring Redis, consult its Laravel documentation page.

Cache Usage

Obtaining A Cache Instance

The `Illuminate\Contracts\Cache\Factory` and `Illuminate\Contracts\Cache\Repository` contracts provide access to Laravel's cache services. The `Factory` contract provides access to all cache drivers defined for your application. The `Repository` contract is typically an implementation of the default cache driver for your application as specified by your `cache` configuration file.

However, you may also use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Laravel cache contracts.

For example, let's import the `Cache` facade into a controller:

```
<?php

namespace App\Http\Controllers;

use Cache;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

Accessing Multiple Cache Stores

Using the `Cache` facade, you may access various cache stores via the `store` method. The key passed to the `store` method should correspond to one of the stores listed in the `stores` configuration array in your `cache` configuration file:

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

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

Retrieving Items From The Cache

The `get` method on the `Cache` facade is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned. If you wish, you may pass a second argument to the `get` method specifying the custom default value you wish to be returned if the item doesn't exist:

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

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

You may even pass a `Closure` as the default value. The result of the `Closure` will be returned if the specified item does not exist in the cache. Passing a Closure allows you to defer the retrieval of default values from a database

or other external service:

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

Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache:

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

Incrementing / Decrementing Values

The `increment` and `decrement` methods may be used to adjust the value of integer items in the cache. Both of these methods optionally accept a second argument indicating the amount by which to increment or decrement the item's value:

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

Cache::increment('key', $amount);

Cache::decrement('key');

Cache::decrement('key', $amount);
```

Retrieve Or Update

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. For example, you may wish to retrieve all users from the cache or, if they don't exist, retrieve them from the database and add them to the cache. You may do this using the `Cache::remember` method:

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

If the item does not exist in the cache, the `Closure` passed to the `remember` method will be executed and its result will be placed in the cache.

You may also combine the `remember` and `forever` methods:

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

Retrieve And Delete

If you need to retrieve an item from the cache and then delete it, you may use the `pull` method. Like the `get` method, `null` will be returned if the item does not exist in the cache:

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

Storing Items In The Cache

You may use the `put` method on the `Cache` facade to store items in the cache. When you place an item in the cache, you will need to specify the number of minutes for which the value should be cached:

```
Cache::put('key', 'value', $minutes);
```

Instead of passing the number of minutes until the item expires, you may also pass a PHP `DateTime` instance representing the expiration time of the cached item:

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

The `add` method will only add the item to the cache if it does not already exist in the cache store. The method will return `true` if the item is actually added to the cache. Otherwise, the method will return `false`:

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

The `forever` method may be used to store an item in the cache permanently. These values must be manually removed from the cache using the `forget` method:

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

Removing Items From The Cache

You may remove items from the cache using the `forget` method on the `Cache` facade:

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

You may clear the entire cache using the `flush` method:

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

Flushing the cache **does not** respect the cache prefix and will remove all entries from the cache. Consider this carefully when clearing a cache which is shared by other applications.

Adding Custom Cache Drivers

To extend the Laravel cache with a custom driver, we will use the `extend` method on the `Cache` facade, which is used to bind a custom driver resolver to the manager. Typically, this is done within a service provider.

For example, to register a new cache driver named "mongo":

```
<?php

namespace App\Providers;

use Cache;
use App\Extensions\MongoStore;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function($app) {
            return Cache::repository(new MongoStore);
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `config/cache.php` configuration file. The second argument is a Closure that should return an `Illuminate\Cache\Repository` instance. The Closure will be passed an `$app` instance, which is an instance of the service container.

The call to `Cache::extend` could be done in the `boot` method of the default `App\Providers\AppServiceProvider` that ships with fresh Laravel applications, or you may create your own service provider to house the extension -

just don't forget to register the provider in the `config/app.php` provider array.

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` contract. So, our MongoDB cache implementation would look something like this:

```
<?php

namespace App\Extensions;

class MongoStore implements \Illuminate\Contracts\Cache\Store
{
    public function get($key) {}
    public function put($key, $value, $minutes) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

We just need to implement each of these methods using a MongoDB connection. Once our implementation is complete, we can finish our custom driver registration:

```
Cache::extend('mongo', function($app) {
    return Cache::repository(new MongoStore);
});
```

Once your extension is complete, simply update your `config/cache.php` configuration file's `driver` option to the name of your extension.

If you're wondering where to put your custom cache driver code, consider making it available on Packagist! Or, you could create an `Extensions` namespace within your `app` directory. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Cache Tags

> **Note:** Cache tags are not supported when using the `file` or `database` cache drivers. Furthermore, when using multiple tags with caches that are stored "forever", performance will be best with a driver such as `memcached`, which automatically purges stale records.

Storing Tagged Cache Items

Cache tags allow you to tag related items in the cache and then flush all cached values that assigned a given tag. You may access a tagged cache by passing in an ordered array of tag names. For example, let's access a tagged cache and `put` value in the cache:

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

However, you are not limited to the `put` method. You may use any cache storage method while working with tags.

Accessing Tagged Cache Items

To retrieve a tagged cache item, pass the same ordered list of tags to the `tags` method:

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

You may flush all items that are assigned a tag or list of tags. For example, this statement would remove all caches tagged with either `people`, `authors`, or both. So, both `Anne` and `John` would be removed from the cache:

```
Cache::tags(['people', 'authors'])->flush();
```

In contrast, this statement would remove only caches tagged with `authors`, so `Anne` would be removed, but not `John`.

```
Cache::tags('authors')->flush();
```

Cache Events

To execute code on every cache operation, you may listen for the events fired by the cache. Typically, you should place these event listeners within the `boot` method of your `EventServiceProvider`:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    $events->listen('cache.hit', function ($key, $value) {
        //
    });

    $events->listen('cache.missed', function ($key) {
        //
    });

    $events->listen('cache.write', function ($key, $value, $minutes) {
        //
    });

    $events->listen('cache.delete', function ($key) {
        //
    });
}
```

Services

Collections

Introduction

The `Illuminate\Support\Collection` class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the `collect` helper to create a new collection instance from the array, run the `strtoupper` function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})
->reject(function ($name) {
    return empty($name);
});
```

As you can see, the `Collection` class allows you to chain its methods to perform fluent mapping and reducing of the underlying array. In general, collections are immutable, meaning every `Collection` method returns an entirely new `Collection` instance.

Creating Collections

As mentioned above, the `collect` helper returns a new `Illuminate\Support\Collection` instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]);
```

The results of Eloquent queries are always returned as `Collection` instances.

Available Methods

For the remainder of this documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulating the underlying array. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary:

all
average
avg
chunk
collapse
contains
count
diff
each
every
except
filter
first
flatMap
flatten
flip
forget
forPage
get
groupBy
has
implode
intersect
isEmpty
keyBy
keys
last
map
max
merge
min
only
pluck
pop
prepend
pull
push
put
random
reduce
reject
reverse
search
shift
shuffle
slice
sort
sortBy
sortByDesc
splice
sum
take
toArray
toJson
transform
unique
values
where
whereLoose
zip

Method Listing

all()

The all method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

average()

Alias for the avg method.

avg()

The avg method returns the average value of a given key:

```
$average = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

chunk()

The chunk method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->toArray();

// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. Imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

collapse()

The collapse method collapses a collection of arrays into a single, flat collection:

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

contains()

The contains method determines whether the collection contains a given item:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');
```

```
// true

$collection->contains('New York');

// false
```

You may also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Finally, you may also pass a callback to the `contains` method to perform your own truth test:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($key, $value) {
    return $value > 5;
});

// false
```

The `contains` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value.

`count()`

The `count` method returns the total number of items in the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

`diff()`

The `diff` method compares the collection against another collection or a plain PHP `array` based on its values. This method will return the values in the original collection that are not present in the given collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

`each()`

The `each` method iterates over the items in the collection and passes each item to a callback:

```
$collection = $collection->each(function ($item, $key) {
    //
});
```

If you would like to stop iterating through the items, you may return `false` from your callback:

```
$collection = $collection->each(function ($item, $key) {
    if (/* some condition */) {
        return false;
    }
});
```

`every()`

The `every` method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->every(4);

// ['a', 'e']
```

You may optionally pass an offset as the second argument:

```
$collection->every(4, 1);

// ['b', 'f']
```

except()

The `except` method returns all items in the collection except for those with the specified keys:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

For the inverse of `except`, see the only method.

filter()

The `filter` method filters the collection using the given callback, keeping only those items that pass a given truth test:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to `false` will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

For the inverse of `filter`, see the reject method.

first()

The `first` method returns the first element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->first(function ($key, $value) {
    return $value > 2;
});

// 3
```

You may also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->first();

// 1
```

flatMap()

The `flatMap` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items. Then, the array is flattened by a level:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

The `flatten` method flattens a multi-dimensional collection into a single dimension:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

flip()

The `flip` method swaps the collection's keys with their corresponding values:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

forget()

The `forget` method removes an item from the collection by its key:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

> **Note:** Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

forPage()

The `forPage` method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

get()

The `get` method returns the item at a given key. If the key does not exist, `null` is returned:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

You may optionally pass a default value as the second argument:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

You may even pass a callback as the default value. The result of the callback will be returned if the specified key does not exist:

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

groupBy()

The `groupBy` method groups the collection's items by a given key:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->toArray();

/*
    [
        'account-x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'account-x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

In addition to passing a string `key`, you may also pass a callback. The callback should return the value you wish to key the group by:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->toArray();

/*
    [
        'x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

has()

The `has` method determines if a given key exists in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);

$collection->has('product');

// true
```

implode()

The `implode` method joins the items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

If the collection contains simple strings or numeric values, simply pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

intersect()

The `intersect` method removes any values from the original collection that are not present in the given `array` or collection. The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

isEmpty()

The `isEmpty` method returns `true` if the collection is empty; otherwise, `false` is returned:

```
collect([])->isEmpty();

// true
```

keyBy()

The `keyBy` method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'desk'],
    ['product_id' => 'prod-200', 'name' => 'chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

keys()

The `keys` method returns all of the collection's keys:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

last()

The `last` method returns the last element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->last(function ($key, $value) {
    return $value < 3;
});

// 2
```

You may also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->last();

// 4
```

map()

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

> **Note:** Like most other collection methods, `map` returns a new collection instance; it does not modify the collection it is called on. If you want to transform the original collection, use the `transform` method.

max()

The `max` method returns the maximum value of a given key:

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();
```

```
// 5
```

merge()

The `merge` method merges the given array or collection with the original collection. If a string key in the given items matches a string key in the original collection, the given items's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given items's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min()

The `min` method returns the minimum value of a given key:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

only()

The `only` method returns the items in the collection with the specified keys:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

For the inverse of `only`, see the except method.

pluck()

The `pluck` method retrieves all of the values for a given key:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

You may also specify how you wish the resulting collection to be keyed:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();
```

```
// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop()

The `pop` method removes and returns the last item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

prepend()

The `prepend` method adds an item to the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to set the key of the prepended item:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

pull()

The `pull` method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

push()

The `push` method appends an item to the end of the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

put()

The `put` method sets the given key and value in the collection:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);
```

```
$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

The `random` method returns a random item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

You may optionally pass an integer to `random` to specify how many items you would like to randomly retrieve. If that integer is more than `1`, a collection of items is returned:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

reduce()

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

The value for `$carry` on the first iteration is `null`; however, you may specify its initial value by passing a second argument to `reduce`:

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

reject()

The `reject` method filters the collection using the given callback. The callback should return `true` if the item should be removed from the resulting collection:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

For the inverse of the `reject` method, see the `filter` method.

reverse()

The `reverse` method reverses the order of the collection's items:

```
$collection = collect([1, 2, 3, 4, 5]);

$reversed = $collection->reverse();

$reversed->all();
```

```
// [5, 4, 3, 2, 1]
```

search()

The `search` method searches the collection for the given value and returns its key if found. If the item is not found, `false` is returned.

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value. To use "strict" comparison, pass `true` as the second argument to the method:

```
$collection->search('4', true);

// false
```

Alternatively, you may pass in your own callback to search for the first item that passes your truth test:

```
$collection->search(function ($item, $key) {
    return $item > 5;
});

// 2
```

shift()

The `shift` method removes and returns the first item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

shuffle()

The `shuffle` method randomly shuffles the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

slice()

The `slice` method returns a slice of the collection starting at the given index:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method:

```
$slice = $collection->slice(4, 2);

$slice->all();
```

```
// [5, 6]
```

The returned slice will have new, numerically indexed keys. If you wish to preserve the original keys, pass `true` as the third argument to the method.

sort()

The `sort` method sorts the collection. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

If your sorting needs are more advanced, you may pass a callback to `sort` with your own algorithm. Refer to the PHP documentation on `usort`, which is what the collection's `sort` method calls under the hood.

If you need to sort a collection of nested arrays or objects, see the `sortBy` and `sortByDesc` methods.

sortBy()

The `sortBy` method sorts the collection by the given key. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

You can also pass your own callback to determine how to sort the collection values:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
    ]
*/
```

sortByDesc()

This method has the same signature as the `sortBy` method, but will sort the collection in the opposite order.

splice()

The `splice` method removes and returns a slice of items starting at the specified index:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

You may pass a second argument to limit the size of the resulting chunk:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

In addition, you can pass a third argument containing the new items to replace the items removed from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

sum()

The `sum` method returns the sum of all items in the collection:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

If the collection contains nested arrays or objects, you should pass a key to use for determining which values to sum:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

In addition, you may pass your own callback to determine which values of the collection to sum:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
```

```
});

// 6
```

take()

The `take` method returns a new collection with the specified number of items:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

You may also pass a negative integer to take the specified amount of items from the end of the collection:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

toArray()

The `toArray` method converts the collection into a plain PHP `array`. If the collection's values are Eloquent models, the models will also be converted to arrays:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

> **Note:** `toArray` also converts all of the collection's nested objects to an array. If you want to get the raw underlying array, use the `all` method instead.

toJson()

The `toJson` method converts the collection into a JSON serialized string:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

transform()

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

> **Note:** Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the `map` method.

unique()

The `unique` method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

When dealing with nested arrays or objects, you may specify the key used to determine uniqueness:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ]
*/
```

You may also pass your own callback to determine item uniqueness:

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
    ]
*/
```

The `unique` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value.

values()

The `values` method returns a new collection with the keys reset to consecutive integers:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);

$values = $collection->values();

$values->all();

/*
    [
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

where()

The `where` method filters the collection by a given key / value pair:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `where` method uses "strict" comparisons when checking item values, meaning a string with an integer value will not be considered equal to an integer of the same value. Use the `whereLoose` method to filter using "loose" comparisons.

`whereLoose()`

This method has the same signature as the `where` method; however, all values are compared using "loose" comparisons.

`zip()`

The `zip` method merges together the values of the given array with the values of the original collection at the corresponding index:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Services

Laravel Elixir

Introduction

Laravel Elixir provides a clean, fluent API for defining basic Gulp tasks for your Laravel application. Elixir supports several common CSS and JavaScript pre-processors, and even testing tools. Using method chaining, Elixir allows you to fluently define your asset pipeline. For example:

```
elixir(function(mix) {
    mix.sass('app.scss')
       .coffee('app.coffee');
});
```

If you've ever been confused about how to get started with Gulp and asset compilation, you will love Laravel Elixir. 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.

Installation & Setup

Installing Node

Before triggering Elixir, you must first ensure that Node.js is installed on your machine.

```
node -v
```

By default, Laravel Homestead includes everything you need; however, if you aren't using Vagrant, then you can easily install Node by visiting their download page.

Gulp

Next, you'll want to pull in Gulp as a global NPM package:

```
npm install --global gulp
```

Laravel Elixir

The only remaining step is to install Elixir! Within a fresh installation of Laravel, you'll find a `package.json` file in the root. Think of this like your `composer.json` file, except it defines Node dependencies instead of PHP. You

may install the dependencies it references by running:

```
npm install
```

If you are developing on a Windows system or you are running your VM on a Windows host system, you may need to run the `npm install` command with the `--no-bin-links` switch enabled:

```
npm install --no-bin-links
```

Running Elixir

Elixir is built on top of Gulp, so to run your Elixir tasks you only need to run the `gulp` command in your terminal. Adding the `--production` flag to the command will instruct Elixir to minify your CSS and JavaScript files:

```
// Run all tasks...
gulp

// Run all tasks and minify all CSS and JavaScript...
gulp --production
```

Watching Assets For Changes

Since it is inconvenient to run the `gulp` command on your terminal after every change to your assets, you may use the `gulp watch` command. This command will continue running in your terminal and watch your assets for any changes. When changes occur, new files will automatically be compiled:

```
gulp watch
```

Working With Stylesheets

The `gulpfile.js` file in your project's root directory contains all of your Elixir tasks. Elixir tasks can be chained together to define exactly how your assets should be compiled.

Less

To compile Less into CSS, you may use the `less` method. The `less` method assumes that your Less files are stored in `resources/assets/less`. By default, the task will place the compiled CSS for this example in `public/css/app.css`:

```
elixir(function(mix) {
    mix.less('app.less');
});
```

You may also combine multiple Less files into a single CSS file. Again, the resulting CSS will be placed in `public/css/app.css`:

```
elixir(function(mix) {
    mix.less([
        'app.less',
        'controllers.less'
    ]);
});
```

If you wish to customize the output location of the compiled CSS, you may pass a second argument to the `less` method:

```
elixir(function(mix) {
    mix.less('app.less', 'public/stylesheets');
});

// Specifying a specific output filename...
elixir(function(mix) {
    mix.less('app.less', 'public/stylesheets/style.css');
});
```

Sass

The `sass` method allows you to compile Sass into CSS. Assuming your Sass files are stored at `resources/assets/sass`, you may use the method like so:

```
elixir(function(mix) {
    mix.sass('app.scss');
});
```

Again, like the `less` method, you may compile multiple Sass files into a single CSS file, and even customize the output directory of the resulting CSS:

```
elixir(function(mix) {
    mix.sass([
        'app.scss',
        'controllers.scss'
    ], 'public/assets/css');
});
```

Plain CSS

If you would just like to combine some plain CSS stylesheets into a single file, you may use the `styles` method. Paths passed to this method are relative to the `resources/assets/css` directory and the resulting CSS will be placed in `public/css/all.css`:

```
elixir(function(mix) {
    mix.styles([
        'normalize.css',
        'main.css'
    ]);
});
```

Of course, you may also output the resulting file to a custom location by passing a second argument to the `styles` method:

```
elixir(function(mix) {
    mix.styles([
        'normalize.css',
        'main.css'
    ], 'public/assets/css');
});
```

Source Maps

Source maps are enabled out of the box. So, for each file that is compiled you will find a companion `*.css.map` file in the same directory. This mapping allows you to trace your compiled stylesheet selectors back to your original Sass or Less while debugging in your browser.

If you do not want source maps generated for your CSS, you may disable them using a simple configuration option:

```
elixir.config.sourcemaps = false;

elixir(function(mix) {
    mix.sass('app.scss');
});
```

Working With Scripts

Elixir also provides several functions to help you work with your JavaScript files, such as compiling ECMAScript 6, compiling CoffeeScript, Browserify, minification, and simply concatenating plain JavaScript files.

CoffeeScript

The `coffee` method may be used to compile CoffeeScript into plain JavaScript. The `coffee` function accepts a

string or array of CoffeeScript files relative to the `resources/assets/coffee` directory and generates a single `app.js` file in the `public/js` directory:

```
elixir(function(mix) {
    mix.coffee(['app.coffee', 'controllers.coffee']);
});
```

Browserify

Elixir also ships with a `browserify` method, which gives you all the benefits of requiring modules in the browser and using ECMAScript 6.

This task assumes that your scripts are stored in `resources/assets/js` and will place the resulting file in `public/js/main.js`:

```
elixir(function(mix) {
    mix.browserify('main.js');
});
```

While Browserify ships with the Partialify and Babelify transformers, you're free to install and add more if you wish:

```
npm install aliasify --save-dev
```

```
elixir.config.js.browserify.transformers.push({
    name: 'aliasify',
    options: {}
});
```

```
elixir(function(mix) {
    mix.browserify('main.js');
});
```

Babel

The `babel` method may be used to compile ECMAScript 6 and 7 into plain JavaScript. This function accepts an array of files relative to the `resources/assets/js` directory, and generates a single `all.js` file in the `public/js` directory:

```
elixir(function(mix) {
    mix.babel([
        'order.js',
        'product.js'
    ]);
});
```

To choose a different output location, simply specify your desired path as the second argument. The signature and functionality of this method are identical to `mix.scripts()`, excluding the Babel compilation.

Scripts

If you have multiple JavaScript files that you would like to combine into a single file, you may use the `scripts` method.

The `scripts` method assumes all paths are relative to the `resources/assets/js` directory, and will place the resulting JavaScript in `public/js/all.js` by default:

```
elixir(function(mix) {
    mix.scripts([
        'jquery.js',
        'app.js'
    ]);
});
```

If you need to combine multiple sets of scripts into different files, you may make multiple calls to the `scripts` method. The second argument given to the method determines the resulting file name for each concatenation:

```
elixir(function(mix) {
```

```
    mix.scripts(['app.js', 'controllers.js'], 'public/js/app.js')
       .scripts(['forum.js', 'threads.js'], 'public/js/forum.js');
});
```

If you need to combine all of the scripts in a given directory, you may use the `scriptsIn` method. The resulting JavaScript will be placed in `public/js/all.js`:

```
elixir(function(mix) {
    mix.scriptsIn('public/js/some/directory');
});
```

Copying Files & Directories

The `copy` method may be used to copy files and directories to new locations. All operations are relative to the project's root directory:

```
elixir(function(mix) {
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');
});

elixir(function(mix) {
    mix.copy('vendor/package/views', 'resources/views');
});
```

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. Elixir can handle this for you using the `version` method.

The `version` method accepts a file name relative to the `public` directory, and will append a unique hash to the filename, allowing for cache-busting. For example, the generated file name will look something like: `all-16d570a7.css`:

```
elixir(function(mix) {
    mix.version('css/all.css');
});
```

After generating the versioned file, you may use Laravel's global `elixir` PHP helper function within your views to load the appropriately hashed asset. The `elixir` function will automatically determine the name of the hashed file:

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">
```

Versioning Multiple Files

You may pass an array to the `version` method to version multiple files:

```
elixir(function(mix) {
    mix.version(['css/all.css', 'js/app.js']);
});
```

Once the files have been versioned, you may use the `elixir` helper function to generate links to the proper hashed files. Remember, you only need to pass the name of the un-hashed file to the `elixir` helper function. The helper will use the un-hashed name to determine the current hashed version of the file:

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">

<script src="{{ elixir('js/app.js') }}"></script>
```

BrowserSync

BrowserSync automatically refreshes your web browser after you make changes to your front-end resources. You can use the `browserSync` method to instruct Elixir to start a BrowserSync server when you run the `gulp watch` command:

```
elixir(function(mix) {
```

```
    mix.browserSync();
});
```

Once you run `gulp watch`, access your web application using port 3000 to enable browser syncing: `http://homestead.app:3000`. If you're using a domain other than `homestead.app` for local development, you may pass an array of options as the first argument to the `browserSync` method:

```
elixir(function(mix) {
    mix.browserSync({
        proxy: 'project.app'
    });
});
```

Calling Existing Gulp Tasks

If you need to call an existing Gulp task from Elixir, you may use the `task` method. As an example, imagine that you have a Gulp task that simply speaks a bit of text when called:

```
gulp.task('speak', function() {
    var message = 'Tea...Earl Grey...Hot';

    gulp.src('').pipe(shell('say ' + message));
});
```

If you wish to call this task from Elixir, use the `mix.task` method and pass the name of the task as the only argument to the method:

```
elixir(function(mix) {
    mix.task('speak');
});
```

Custom Watchers

If you need to register a watcher to run your custom task each time some files are modified, pass a regular expression as the second argument to the `task` method:

```
elixir(function(mix) {
    mix.task('speak', 'app/**/*.php');
});
```

Writing Elixir Extensions

If you need more flexibility than Elixir's `task` method can provide, you may create custom Elixir extensions. Elixir extensions allow you to pass arguments to your custom tasks. For example, you could write an extension like so:

```
// File: elixir-extensions.js

var gulp = require('gulp');
var shell = require('gulp-shell');
var Elixir = require('laravel-elixir');

var Task = Elixir.Task;

Elixir.extend('speak', function(message) {

    new Task('speak', function() {
        return gulp.src('').pipe(shell('say ' + message));
    });

});

// mix.speak('Hello World');
```

That's it! Notice that your Gulp-specific logic should be placed within the function passed as the second argument to the `Task` constructor. You may either place this at the top of your Gulpfile, or instead extract it to a custom tasks file. For example, if you place your extensions in `elixir-extensions.js`, you may require the file from your main `Gulpfile` like so:

```
// File: Gulpfile.js

var elixir = require('laravel-elixir');

require('./elixir-extensions')

elixir(function(mix) {
    mix.speak('Tea, Earl Grey, Hot');
});
```

Custom Watchers

If you would like your custom task to be re-triggered while running `gulp watch`, you may register a watcher:

```
new Task('speak', function() {
    return gulp.src('').pipe(shell('say ' + message));
})
.watch('./app/**');
```

Services

Encryption

Configuration

Before using Laravel's encrypter, you should set the `key` option of your `config/app.php` configuration file to a 32 character, random string. If this value is not properly set, all values encrypted by Laravel will be insecure.

Basic Usage

Encrypting A Value

You may encrypt a value using the `Crypt` facade. All encrypted values are encrypted using OpenSSL and the `AES-256-CBC` cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC) to detect any modifications to the encrypted string.

For example, we may use the `encrypt` method to encrypt a secret and store it on an Eloquent model:

```
<?php

namespace App\Http\Controllers;

use Crypt;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Store a secret message for the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => Crypt::encrypt($request->secret)
        ])->save();
    }
}
```

Decrypting A Value

Of course, you may decrypt values using the `decrypt` method on the `Crypt` facade. If the value can not be properly decrypted, such as when the MAC is invalid, an `Illuminate\Contracts\Encryption\DecryptException` will be thrown:

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = Crypt::decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

Services

Errors & Logging

Introduction

When you start a new Laravel project, error and exception handling is already configured for you. In addition, Laravel is integrated with the Monolog logging library, which provides support for a variety of powerful log handlers.

Configuration

Error Detail

The amount of error detail your application displays through the browser is controlled by the `debug` configuration option in your `config/app.php` configuration file. By default, this configuration option is set to respect the `APP_DEBUG` environment variable, which is stored in your `.env` file.

For local development, you should set the `APP_DEBUG` environment variable to `true`. In your production environment, this value should always be `false`.

Log Modes

Out of the box, Laravel supports `single`, `daily`, `syslog` and `errorlog` logging modes. For example, if you wish to use daily log files instead of a single file, you should simply set the `log` value in your `config/app.php` configuration file:

```
'log' => 'daily'
```

Custom Monolog Configuration

If you would like to have complete control over how Monolog is configured for your application, you may use the application's `configureMonologUsing` method. You should place a call to this method in your `bootstrap/app.php` file right before the `$app` variable is returned by the file:

```
$app->configureMonologUsing(function($monolog) {
    $monolog->pushHandler(...);
});

return $app;
```

The Exception Handler

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`. We'll examine each of these methods in detail.

The Report Method

The `report` method is used to log exceptions or send them to an external service like BugSnag. By default, the `report` method simply passes the exception to the base class where the exception is logged. However, you are

free to log exceptions however you wish.

For example, if you need to report different types of exceptions in different ways, you may use the PHP `instanceof` comparison operator:

```
/**
 * Report or log an exception.
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param  \Exception  $e
 * @return void
 */
public function report(Exception $e)
{
    if ($e instanceof CustomException) {
        //
    }

    return parent::report($e);
}
```

Ignoring Exceptions By Type

The `$dontReport` property of the exception handler contains an array of exception types that will not be logged. By default, exceptions resulting from 404 errors are not written to your log files. You may add other exception types to this array as needed.

The Render Method

The `render` method is responsible for converting a given exception into an HTTP response that should be sent back to the browser. By default, the exception is passed to the base class which generates a response for you. However, you are free to check the exception type or return your own custom response:

```
/**
 * Render an exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $e
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e)
{
    if ($e instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $e);
}
```

HTTP Exceptions

Some exceptions describe HTTP error codes from the server. For example, this may be a "page not found" error (404), an "unauthorized error" (401) or even a developer generated 500 error. In order to generate such a response from anywhere in your application, use the following:

```
abort(404);
```

The `abort` method will immediately raise an exception which will be rendered by the exception handler. Optionally, you may provide the response text:

```
abort(403, 'Unauthorized action.');
```

This method may be used at any time during the request's lifecycle.

Custom HTTP Error Pages

Laravel makes it easy to return custom error pages for various HTTP status codes. For example, if you wish to

customize the error page for 404 HTTP status codes, create a `resources/views/errors/404.blade.php`. This file will be served on all 404 errors generated by your application.

The views within this directory should be named to match the HTTP status code they correspond to.

Logging

The Laravel logging facilities provide a simple layer on top of the powerful Monolog library. By default, Laravel is configured to create daily log files for your application which are stored in the `storage/logs` directory. You may write information to the logs using the `Log` facade:

```
<?php

namespace App\Http\Controllers;

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

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

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

The logger provides the eight logging levels defined in RFC 5424: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info** and **debug**.

```
Log::emergency($error);
Log::alert($error);
Log::critical($error);
Log::error($error);
Log::warning($error);
Log::notice($error);
Log::info($error);
Log::debug($error);
```

Contextual Information

An array of contextual data may also be passed to the log methods. This contextual data will be formatted and displayed with the log message:

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

Accessing The Underlying Monolog Instance

Monolog has a variety of additional handlers you may use for logging. If needed, you may access the underlying Monolog instance being used by Laravel:

```
$monolog = Log::getMonolog();
```

Services

Events

Introduction

Laravel's events provides a simple observer implementation, allowing you to subscribe and listen for events in your application. Event classes are typically stored in the `app/Events` directory, while their listeners are stored in `app/Listeners`.

Registering Events / Listeners

The `EventServiceProvider` included with your Laravel application provides a convenient place to register all event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). Of course, you may add as many events to this array as your application requires. For example, let's add our `PodcastWasPurchased` event:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Listeners\EmailPurchaseConfirmation',
    ],
];
```

Generating Event / Listener Classes

Of course, manually creating the files for each event and listener is cumbersome. Instead, simply add listeners and events to your `EventServiceProvider` and use the `event:generate` command. This command will generate any events or listeners that are listed in your `EventServiceProvider`. Of course, events and listeners that already exist will be left untouched:

```
php artisan event:generate
```

Registering Events Manually

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register events manually with the event dispatcher using either the `Event` facade or the `Illuminate\Contracts\Events\Dispatcher` contract implementation:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
```

```
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    $events->listen('event.name', function ($foo, $bar) {
        //
    });
}
```

Wildcard Event Listeners

You may even register listeners using the `*` as a wildcard, allowing you to catch multiple events on the same listener. Wildcard listeners receive the entire event data array as a single argument:

```
$events->listen('event.*', function (array $data) {
    //
});
```

Defining Events

An event class is simply a data container which holds the information related to the event. For example, let's assume our generated `PodcastWasPurchased` event receives an Eloquent ORM object:

```
<?php

namespace App\Events;

use App\Podcast;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class PodcastWasPurchased extends Event
{
    use SerializesModels;

    public $podcast;

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

As you can see, this event class contains no special logic. It is simply a container for the `Podcast` object that was purchased. The `SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's `serialize` function.

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive the event instance in their `handle` method. The `event:generate` command will automatically import the proper event class and type-hint the event on the `handle` method. Within the `handle` method, you may perform any logic necessary to respond to the event.

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation
{
```

```
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  PodcastWasPurchased  $event
     * @return void
     */
    public function handle(PodcastWasPurchased $event)
    {
        // Access the podcast using $event->podcast...
    }
}
```

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:

```
use Illuminate\Contracts\Mail\Mailer;

public function __construct(Mailer $mailer)
{
    $this->mailer = $mailer;
}
```

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

Need to queue an event listener? It couldn't be any easier. Simply add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` Artisan command already have this interface imported into the current namespace, so you can use it immediately:

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue
{
    //
}
```

That's it! Now, when this listener is called for an event, it will be queued automatically 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 processed.

Manually Accessing The Queue

If you need to access the underlying queue job's `delete` and `release` methods manually, you may do so. The `Illuminate\Queue\InteractsWithQueue` trait, which is imported by default on generated listeners, gives you access to these methods:

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
```

```
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(PodcastWasPurchased $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

Firing Events

To fire an event, you may use the `Event` facade, passing an instance of the event to the `fire` method. The `fire` method will dispatch the event to all of its registered listeners:

```
<?php

namespace App\Http\Controllers;

use Event;
use App\Podcast;
use App\Events\PodcastWasPurchased;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $userId
     * @param  int  $podcastId
     * @return Response
     */
    public function purchasePodcast($userId, $podcastId)
    {
        $podcast = Podcast::findOrFail($podcastId);

        // Purchase podcast logic...

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

Alternatively, you may use the global `event` helper function to fire events:

```
event(new PodcastWasPurchased($podcast));
```

Broadcasting Events

In many modern web applications, web sockets are used to implement real-time, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a websocket connection to be handled by the client.

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

Configuration

All of the event broadcasting configuration options are stored in the `config/broadcasting.php` configuration file. Laravel supports several broadcast drivers out of the box: Pusher, Redis, and a `log` driver for local development and debugging. A configuration example is included for each of these drivers.

Broadcast Prerequisites

The following dependencies are needed for event broadcasting:

- Pusher: `pusher/pusher-php-server ~2.0`
- Redis: `predis/predis ~1.0`

Queue Prerequisites

Before broadcasting events, you will also need to configure and run a queue listener. All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected.

Marking Events For Broadcast

To inform Laravel that a given event should be broadcast, implement the `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface on the event class. The `ShouldBroadcast` interface requires you to implement a single method: `broadcastOn`. The `broadcastOn` method should return an array of "channel" names that the event should be broadcast on:

```
<?php

namespace App\Events;

use App\User;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated extends Event implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

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

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

Then, you only need to fire the event as you normally would. Once the event has been fired, a queued job will automatically broadcast the event over your specified broadcast driver.

Overriding Broadcast Event Name

By default, the broadcast event name will be the fully qualified class name of the event. Using the example class above, the broadcast event would be `App\Events\ServerCreated`. You can customize this broadcast event name to whatever you want using the `broadcastAs` method:

```
/**
 * Get the broadcast event name.
 *
 * @return string
 */
public function broadcastAs()
{
    return 'app.server-created';
}
```

Broadcast Data

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

```
{
    "user": {
        "id": 1,
        "name": "Jonathan Banks"
        ...
    }
}
```

However, if you wish to have even 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 with the event:

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

Consuming Event Broadcasts

Pusher

You may conveniently consume events broadcast using the Pusher driver using Pusher's JavaScript SDK. For example, let's consume the `App\Events\ServerCreated` event from our previous examples:

```
this.pusher = new Pusher('pusher-key');

this.pusherChannel = this.pusher.subscribe('user.' + USER_ID);

this.pusherChannel.bind('App\\Events\\ServerCreated', function(message) {
    console.log(message.user);
});
```

Redis

If you are using the Redis broadcaster, you will need to write your own Redis pub/sub consumer to receive the messages and broadcast them using the websocket technology of your choice. For example, you may choose to use the popular Socket.io library which is written in Node.

Using the `socket.io` and `ioredis` Node libraries, you can quickly write an event broadcaster to publish all events that are broadcast by your Laravel application:

```
var app = require('http').createServer(handler);
var io = require('socket.io')(app);

var Redis = require('ioredis');
var redis = new Redis();

app.listen(6001, function() {
    console.log('Server is running!');
});

function handler(req, res) {
    res.writeHead(200);
    res.end('');
}

io.on('connection', function(socket) {
    //
});
```

```
redis.psubscribe('*', function(err, count) {
    //
});

redis.on('pmessage', function(subscribed, channel, message) {
    message = JSON.parse(message);
    io.emit(channel + ':' + message.event, message.data);
});
```

Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the class itself, allowing you to define several event handlers within a single class. Subscribers should define a `subscribe` method, which will be passed an event dispatcher instance:

```
<?php

namespace App\Listeners;

class UserEventListener
{
    /**
     * Handle user login events.
     */
    public function onUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function onUserLogout($event) {}

    /**
     * Register the listeners for the subscriber.
     *
     * @param  Illuminate\Events\Dispatcher  $events
     */
    public function subscribe($events)
    {
        $events->listen(
            'App\Events\UserLoggedIn',
            'App\Listeners\UserEventListener@onUserLogin'
        );

        $events->listen(
            'App\Events\UserLoggedOut',
            'App\Listeners\UserEventListener@onUserLogout'
        );
    }

}
```

Registering An Event Subscriber

Once the subscriber has been defined, it may be registered with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventListener`.

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Events\Dispatcher as DispatcherContract;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];
```

```
    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventListener',
    ];
}
```

Framework Events

Laravel provides a variety of "core" events for actions performed by the framework. You can subscribe to them in the same way that you subscribe to your own custom events:

Event	Parameter(s)
artisan.start	$application
auth.attempt	$credentials, $remember, $login
auth.login	$user, $remember
auth.logout	$user
cache.missed	$key
cache.hit	$key, $value
cache.write	$key, $value, $minutes
cache.delete	$key
connection.{name}.beganTransaction	$connection
connection.{name}.committed	$connection
connection.{name}.rollingBack	$connection
illuminate.query	$query, $bindings, $time, $connectionName
illuminate.queue.after	$connection, $job, $data
illuminate.queue.failed	$connection, $job, $data
illuminate.queue.stopping	null
mailer.sending	$message
router.matched	$route, $request
composing:{view name}	$view
creating:{view name}	$view

Services

Filesystem / Cloud Storage

Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple to use drivers for working with local filesystems, Amazon S3, and Rackspace Cloud Storage. Even better, it's amazingly simple to switch between these storage options as the API remains the same for each system.

Configuration

The filesystem configuration file is located at `config/filesystems.php`. Within this file you may configure all of your "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver is included in the configuration file. So, simply modify the configuration to reflect your storage preferences and credentials.

Of course, you may configure as many disks as you like, and may even have multiple disks that use the same driver.

The Local Driver

When using the `local` driver, note that all file operations are relative to the `root` directory defined in your configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would store a file in `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Other Driver Prerequisites

Before using the S3 or Rackspace drivers, you will need to install the appropriate package via Composer:

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

Basic Usage

Obtaining Disk Instances

The `Storage` facade may be used to interact with any of your configured disks. For example, you may use the `put` method on the facade to store an avatar on the default disk. If you call methods on the `Storage` facade without first calling the `disk` method, the method call will automatically be passed to the default disk:

```
<?php

namespace App\Http\Controllers;
```

```
use Storage;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Update the avatar for the given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function updateAvatar(Request $request, $id)
    {
        $user = User::findOrFail($id);

        Storage::put(
            'avatars/'.$user->id,
            file_get_contents($request->file('avatar')->getRealPath())
        );
    }
}
```

When using multiple disks, you may access a particular disk using the `disk` method on the `Storage` facade. Of course, you may continue to chain methods to execute methods on the disk:

```
$disk = Storage::disk('s3');

$contents = Storage::disk('local')->get('file.jpg');
```

Retrieving Files

The `get` method may be used to retrieve the contents of a given file. The raw string contents of the file will be returned by the method:

```
$contents = Storage::get('file.jpg');
```

The `has` method may be used to determine if a given file exists on the disk:

```
$exists = Storage::disk('s3')->has('file.jpg');
```

File Meta Information

The `size` method may be used to get the size of the file in bytes:

```
$size = Storage::size('file1.jpg');
```

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file1.jpg');
```

Storing Files

The `put` method may be used to store a file on disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Using streams is greatly recommended when dealing with large files:

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

Storage::put('file.jpg', $resource);
```

The `copy` method may be used to copy an existing file to a new location on the disk:

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

The `move` method may be used to rename or move an existing file to a new location:

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Prepending / Appending To Files

The `prepend` and `append` methods allow you to easily insert content at the beginning or end of a file:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Deleting Files

The `delete` method accepts a single filename or an array of files to remove from the disk:

```
Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

Directories

Get All Files Within A Directory

The `files` method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all sub-directories, you may use the `allFiles` method:

```
$files = Storage::files($directory);

$files = Storage::allFiles($directory);
```

Get All Directories Within A Directory

The `directories` method returns an array of all the directories within a given directory. Additionally, you may use the `allDirectories` method to get a list of all directories within a given directory and all of its sub-directories:

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

Create A Directory

The `makeDirectory` method will create the given directory, including any needed sub-directories:

```
Storage::makeDirectory($directory);
```

Delete A Directory

Finally, the `deleteDirectory` may be used to remove a directory, including all of its files, from the disk:

```
Storage::deleteDirectory($directory);
```

Custom Filesystems

Laravel's Flysystem integration provides drivers for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application.

In order to set up the custom filesystem you will need to create a service provider such as `DropboxServiceProvider`. In the provider's `boot` method, you may use the `Storage` facade's `extend` method to define the custom driver:

```
<?php

namespace App\Providers;
```

```
use Storage;
use League\Flysystem\Filesystem;
use Dropbox\Client as DropboxClient;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Dropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function($app, $config) {
            $client = new DropboxClient(
                $config['accessToken'], $config['clientIdentifier']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

The first argument of the `extend` method is the name of the driver and the second is a Closure that receives the `$app` and `$config` variables. The resolver Closure must return an instance of `League\Flysystem\Filesystem`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Once you have created the service provider to register the extension, you may use the `dropbox` driver in your `config/filesystem.php` configuration file.

Services

Hashing

Introduction

The Laravel `Hash` facade provides secure Bcrypt hashing for storing user passwords. If you are using the `AuthController` controller that is included with your Laravel application, it will automatically use Bcrypt for registration and authentication.

Bcrypt is a great choice for hashing passwords because its "work factor" is adjustable, which means that the time it takes to generate a hash can be increased as hardware power increases.

Basic Usage

You may hash a password by calling the `make` method on the `Hash` facade:

```
<?php

namespace App\Http\Controllers;

use Hash;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function updatePassword(Request $request, $id)
    {
        $user = User::findOrFail($id);

        // Validate the new password length...

        $user->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();
    }
}
```

Alternatively, you may also use the global `bcrypt` helper function:

```
bcrypt('plain-text');
```

Verifying A Password Against A Hash

The `check` method allows you to verify that a given plain-text string corresponds to a given hash. However, if you are using the `AuthController` included with Laravel, you will probably not need to use this directly, as the included authentication controller automatically calls this method:

```
if (Hash::check('plain-text', $hashedPassword)) {
    // The passwords match...
}
```

Checking If A Password Needs To Be Rehashed

The `needsRehash` function allows you to determine if the work factor used by the hasher has changed since the

password was hashed:

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

Services

Helper Functions

Introduction

Laravel includes a variety of "helper" PHP functions. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

Available Methods

Arrays

array_add
array_collapse
array_divide
array_dot
array_except
array_first
array_flatten
array_forget
array_get
array_has
array_only
array_pluck
array_pull
array_set
array_sort
array_sort_recursive
array_where
head
last

Paths

app_path
base_path
config_path
database_path
elixir
public_path
storage_path

Strings

camel_case
class_basename
e
ends_with
snake_case
str_limit
starts_with
str_contains
str_finish
str_is
str_plural
str_random
str_singular
str_slug
studly_case
trans
trans_choice

URLs

action
asset
secure_asset
route
url

Miscellaneous

auth
back
bcrypt
collect
config
csrf_field
csrf_token
dd
env
event
factory
method_field
old
redirect
request
response
session
value
view
with

Method Listing

Arrays

array_add()

The `array_add` function adds a given key / value pair to the array if the given key doesn't already exist in the array:

```
$array = array_add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

array_collapse()

The `array_collapse` function collapse an array of arrays into a single array:

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

array_divide()

The `array_divide` function returns two arrays, one containing the keys, and the other containing the values of the original array:

```
list($keys, $values) = array_divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

array_dot()

The `array_dot` function flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth:

```
$array = array_dot(['foo' => ['bar' => 'baz']]);

// ['foo.bar' => 'baz'];
```

array_except()

The `array_except` function removes the given key / value pairs from the array:

```
$array = ['name' => 'Desk', 'price' => 100];

$array = array_except($array, ['price']);

// ['name' => 'Desk']
```

array_first()

The `array_first` function returns the first element of an array passing a given truth test:

```
$array = [100, 200, 300];

$value = array_first($array, function ($key, $value) {
    return $value >= 150;
});

// 200
```

A default value may also be passed as the third parameter to the method. This value will be returned if no value passes the truth test:

```
$value = array_first($array, $callback, $default);
```

array_flatten()

The `array_flatten` function will flatten a multi-dimensional array into a single level.

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$array = array_flatten($array);

// ['Joe', 'PHP', 'Ruby'];
```

array_forget()

The `array_forget` function removes a given key / value pair from a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

array_forget($array, 'products.desk');

// ['products' => []]
```

array_get()

The `array_get` function retrieves a value from a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

$value = array_get($array, 'products.desk');

// ['price' => 100]
```

The `array_get` function also accepts a default value, which will be returned if the specific key is not found:

```
$value = array_get($array, 'names.john', 'default');
```

array_has()

The `array_has` function checks that a given item exists in an array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

$hasDesk = array_has($array, 'products.desk');

// true
```

array_only()

The `array_only` function will return only the specified key / value pairs from the given array:

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$array = array_only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

array_pluck()

The `array_pluck` function will pluck a list of the given key / value pairs from the array:

```
$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$array = array_pluck($array, 'developer.name');

// ['Taylor', 'Abigail'];
```

You may also specify how you wish the resulting list to be keyed:

```
$array = array_pluck($array, 'developer.name', 'developer.id');
```

```
// [1 => 'Taylor', 2 => 'Abigail'];
```

array_pull()

The `array_pull` function returns and removes a key / value pair from the array:

```
$array = ['name' => 'Desk', 'price' => 100];

$name = array_pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

array_set()

The `array_set` function sets a value within a deeply nested array using "dot" notation:

```
$array = ['products' => ['desk' => ['price' => 100]]];

array_set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

array_sort()

The `array_sort` function sorts the array by the results of the given Closure:

```
$array = [
    ['name' => 'Desk'],
    ['name' => 'Chair'],
];

$array = array_values(array_sort($array, function ($value) {
    return $value['name'];
}));

/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
    ]
*/
```

array_sort_recursive()

The `array_sort_recursive` function recursively sorts the array using the `sort` function:

```
$array = [
    [
        'Roman',
        'Taylor',
        'Li',
    ],
    [
        'PHP',
        'Ruby',
        'JavaScript',
    ],
];

$array = array_sort_recursive($array);

/*
    [
        [
            'Li',
            'Roman',
            'Taylor',
        ],
        [
            'JavaScript',
            'PHP',
            'Ruby',
        ]
```

```
    ];
*/
```

array_where()

The `array_where` function filters the array using the given Closure:

```
$array = [100, '200', 300, '400', 500];

$array = array_where($array, function ($key, $value) {
    return is_string($value);
});

// [1 => 200, 3 => 400]
```

head()

The `head` function simply returns the first element in the given array:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

last()

The `last` function returns the last element in the given array:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

Paths

app_path()

The `app_path` function returns the fully qualified path to the `app` directory:

```
$path = app_path();
```

You may also use the `app_path` function to generate a fully qualified path to a given file relative to the application directory:

```
$path = app_path('Http/Controllers/Controller.php');
```

base_path()

The `base_path` function returns the fully qualified path to the project root:

```
$path = base_path();
```

You may also use the `base_path` function to generate a fully qualified path to a given file relative to the application directory:

```
$path = base_path('vendor/bin');
```

config_path()

The `config_path` function returns the fully qualified path to the application configuration directory:

```
$path = config_path();
```

database_path()

The `database_path` function returns the fully qualified path to the application's database directory:

```
$path = database_path();
```

`elixir()`

The `elixir` function gets the path to the versioned Elixir file:

```
elixir($file);
```

`public_path()`

The `public_path` function returns the fully qualified path to the `public` directory:

```
$path = public_path();
```

`storage_path()`

The `storage_path` function returns the fully qualified path to the `storage` directory:

```
$path = storage_path();
```

You may also use the `storage_path` function to generate a fully qualified path to a given file relative to the storage directory:

```
$path = storage_path('app/file.txt');
```

Strings

`camel_case()`

The `camel_case` function converts the given string to `camelCase`:

```
$camel = camel_case('foo_bar');

// fooBar
```

`class_basename()`

The `class_basename` returns the class name of the given class with the class' namespace removed:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

`e()`

The `e` function runs `htmlentities` over the given string:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

`ends_with()`

The `ends_with` function determines if the given string ends with the given value:

```
$value = ends_with('This is my name', 'name');

// true
```

`snake_case()`

The `snake_case` function converts the given string to `snake_case`:

```
$snake = snake_case('fooBar');
```

```
// foo_bar
```

str_limit()

The `str_limit` function limits the number of characters in a string. The function accepts a string as its first argument and the maximum number of resulting characters as its second argument:

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with()

The `starts_with` function determines if the given string begins with the given value:

```
$value = starts_with('This is my name', 'This');

// true
```

str_contains()

The `str_contains` function determines if the given string contains the given value:

```
$value = str_contains('This is my name', 'my');

// true
```

str_finish()

The `str_finish` function adds a single instance of the given value to a string:

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is()

The `str_is` function determines if a given string matches a given pattern. Asterisks may be used to indicate wildcards:

```
$value = str_is('foo*', 'foobar');

// true

$value = str_is('baz*', 'foobar');

// false
```

str_plural()

The `str_plural` function converts a string to its plural form. This function currently only supports the English language:

```
$plural = str_plural('car');

// cars

$plural = str_plural('child');

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
$plural = str_plural('child', 2);

// children

$plural = str_plural('child', 1);
```

```
// child
```

str_random()

The `str_random` function generates a random string of the specified length:

```
$string = str_random(40);
```

str_singular()

The `str_singular` function converts a string to its singular form. This function currently only supports the English language:

```
$singular = str_singular('cars');

// car
```

str_slug()

The `str_slug` function generates a URL friendly "slug" from the given string:

```
$title = str_slug("Laravel 5 Framework", "-");

// laravel-5-framework
```

studly_case()

The `studly_case` function converts the given string to `StudlyCase`:

```
$value = studly_case('foo_bar');

// FooBar
```

trans()

The `trans` function translates the given language line using your localization files:

```
echo trans('validation.required'):
```

trans_choice()

The `trans_choice` function translates the given language line with inflection:

```
$value = trans_choice('foo.bar', $count);
```

URLs

action()

The `action` function generates a URL for the given controller action. You do not need to pass the full namespace to the controller. Instead, pass the controller class name relative to the `App\Http\Controllers` namespace:

```
$url = action('HomeController@getIndex');
```

If the method accepts route parameters, you may pass them as the second argument to the method:

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

asset()

Generate a URL for an asset using the current scheme of the request (HTTP or HTTPS):

```
$url = asset('img/photo.jpg');
```

secure_asset()

Generate a URL for an asset using HTTPS:

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

route()

The `route` function generates a URL for the given named route:

```
$url = route('routeName');
```

If the route accepts parameters, you may pass them as the second argument to the method:

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

url()

The `url` function generates a fully qualified URL to the given path:

```
echo url('user/profile');

echo url('user/profile', [1]);
```

Miscellaneous

auth()

The `auth` function returns an authenticator instance. You may use it instead of the `Auth` facade for convenience:

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

back()

The `back()` function generates a redirect response to the user's previous location:

```
return back();
```

bcrypt()

The `bcrypt` function hashes the given value using Bcrypt. You may use it as an alternative to the `Hash` facade:

```
$password = bcrypt('my-secret-password');
```

collect()

The `collect` function creates a collection instance from the supplied items:

```
$collection = collect(['taylor', 'abigail']);
```

config()

The `config` function gets the value of a configuration variable. The configuration values may be accessed using "dot" syntax, which includes the name of the file and the option you wish to access. A default value may be specified and is returned if the configuration option does not exist:

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

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

The `config` helper may also be used to set configuration variables at runtime by passing an array of key / value pairs:

```
config(['app.debug' => true]);
```

csrf_field()

The csrf_field function generates an HTML hidden input field containing the value of the CSRF token. For example, using Blade syntax:

```
{!! csrf_field() !!}
```

csrf_token()

The csrf_token function retrieves the value of the current CSRF token:

```
$token = csrf_token();
```

dd()

The dd function dumps the given variable and ends execution of the script:

```
dd($value);
```

env()

The env function gets the value of an environment variable or returns a default value:

```
$env = env('APP_ENV');

// Return a default value if the variable doesn't exist...
$env = env('APP_ENV', 'production');
```

event()

The event function dispatches the given event to its listeners:

```
event(new UserRegistered($user));
```

factory()

The factory function creates a model factory builder for a given class, name, and amount. It can be used while testing or seeding:

```
$user = factory(App\User::class)->make();
```

method_field()

The method_field function generates an HTML hidden input field containing the spoofed value of the form's HTTP verb. For example, using Blade syntax:

```
<form method="POST">
    {!! method_field('delete') !!}
</form>
```

old()

The old function retrieves an old input value flashed into the session:

```
$value = old('value');
```

redirect()

The redirect function returns an instance of the redirector to do redirects:

```
return redirect('/home');
```

request()

The request function returns the current request instance or obtains an input item:

```
$request = request();

$value = request('key', $default = null)
```

response()

The `response` function creates a response instance or obtains an instance of the response factory:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

session()

The `session` function may be used to get / set a session value:

```
$value = session('key');
```

You may set values by passing an array of key / value pairs to the function:

```
session(['chairs' => 7, 'instruments' => 3]);
```

The session store will be returned if no value is passed to the function:

```
$value = session()->get('key');

session()->put('key', $value);
```

value()

The `value` function's behavior will simply return the value it is given. However, if you pass a `Closure` to the function, the `Closure` will be executed then its result will be returned:

```
$value = value(function() { return 'bar'; });
```

view()

The `view` function retrieves a view instance:

```
return view('auth.login');
```

with()

The `with` function returns the value it is given. This function is primarily useful for method chaining where it would otherwise be impossible:

```
$value = with(new Foo)->work();
```

Services

Localization

Introduction

Laravel's localization features provide a convenient way to retrieve strings in various languages, allowing you to easily support multiple languages within your application.

Language strings are stored in files within the `resources/lang` directory. Within this directory there should be a subdirectory for each language supported by the application:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files simply return an array of keyed strings. For example:

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

Configuring The Locale

The default language for your application is stored in the `config/app.php` configuration file. Of course, you may modify this value to suit the needs of your application. You may also change the active language at runtime using the `setLocale` method on the `App` facade:

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);

    //
});
```

You may also configure a "fallback language", which will be used when the active language does not contain a given language line. Like the default language, the fallback language is also configured in the `config/app.php` configuration file:

```
'fallback_locale' => 'en',
```

Basic Usage

You may retrieve lines from language files using the `trans` helper function. The `trans` method accepts the file and key of the language line as its first argument. For example, let's retrieve the language line `welcome` in the `resources/lang/messages.php` language file:

```
echo trans('messages.welcome');
```

Of course if you are using the Blade templating engine, you may use the `{{ }}` syntax to echo the language line:

```
{{ trans('messages.welcome') }}
```

If the specified language line does not exist, the `trans` function will simply return the language line key. So, using the example above, the `trans` function would return `messages.welcome` if the language line does not exist.

Replacing Parameters In Language Lines

If you wish, you may define place-holders in your language lines. All place-holders are prefixed with a `:`. For example, you may define a welcome message with a place-holder name:

```
'welcome' => 'Welcome, :name',
```

To replace the place-holders when retrieving a language line, pass an array of replacements as the second argument to the `trans` function:

```
echo trans('messages.welcome', ['name' => 'Dayle']);
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization. By using a "pipe" character, you may distinguish a singular and plural form of a string:

```
'apples' => 'There is one apple|There are many apples',
```

Then, you may then 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 language line is returned:

```
echo trans_choice('messages.apples', 10);
```

Since the Laravel translator is powered by the Symfony Translation component, you may create even more complex pluralization rules:

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

Overriding Vendor Language Files

Some packages may ship with their own language files. Instead of hacking the package's core files to tweak these lines, you may override them by placing your own files in the `resources/lang/vendor/{package}/{locale}` directory.

So, for example, if you need to override the English language lines in `messages.php` for a package named `skyrim/hearthfire`, you would place a language file at: `resources/lang/vendor/hearthfire/en/messages.php`. In this file you should only define the language lines you wish to override. Any language lines you don't override will still be loaded from the package's original language files.

Services

Mail

Introduction

Laravel provides a clean, simple API over the popular SwiftMailer library. Laravel provides drivers for SMTP, Mailgun, Mandrill, Amazon SES, PHP's `mail` function, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Driver Prerequisites

The API based drivers such as Mailgun and Mandrill are often simpler and faster than SMTP servers. All of the API drivers require that the Guzzle HTTP library be installed for your application. You may install Guzzle to your project by adding the following line to your `composer.json` file:

```
"guzzlehttp/guzzle": "~5.3|~6.0"
```

Mailgun Driver

To use the Mailgun driver, first install Guzzle, then set the `driver` option in your `config/mail.php` configuration file to `mailgun`. Next, verify that your `config/services.php` configuration file contains the following options:

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

Mandrill Driver

To use the Mandrill driver, first install Guzzle, then set the `driver` option in your `config/mail.php` configuration file to `mandrill`. Next, verify that your `config/services.php` configuration file contains the following options:

```
'mandrill' => [
    'secret' => 'your-mandrill-key',
],
```

SES Driver

To use the Amazon SES driver, install the Amazon AWS SDK for PHP. You may install this library by adding the following line to your `composer.json` file's `require` section:

```
"aws/aws-sdk-php": "~3.0"
```

Next, set the `driver` option in your `config/mail.php` configuration file to `ses`. Then, verify that your `config/services.php` configuration file contains the following options:

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region',  // e.g. us-east-1
],
```

Sending Mail

Laravel allows you to store your e-mail messages in views. For example, to organize your e-mails, you could create an `emails` directory within your `resources/views` directory:

To send a message, use the `send` method on the `Mail` facade. The `send` method accepts three arguments. First, the name of a view that contains the e-mail message. Secondly, an array of data you wish to pass to the view. Lastly, a `Closure` callback which receives a message instance, allowing you to customize the recipients, subject, and other aspects of the mail message:

```
<?php

namespace App\Http\Controllers;

use Mail;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Send an e-mail reminder to the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendEmailReminder(Request $request, $id)
    {
        $user = User::findOrFail($id);

        Mail::send('emails.reminder', ['user' => $user], function ($m) use ($user) {
            $m->from('hello@app.com', 'Your Application');

            $m->to($user->email, $user->name)->subject('Your Reminder!');
        });
    }
}
```

Since we are passing an array containing the `user` key in the example above, we could display the user's name within our e-mail view using the following PHP code:

```
<?php echo $user->name; ?>
```

> **Note:** A `$message` variable is always passed to e-mail views, and allows the inline embedding of attachments. So, you should avoid passing a `message` variable in your view payload.

Building The Message

As previously discussed, the third argument given to the `send` method is a `Closure` allowing you to specify various options on the e-mail message itself. Using this Closure you may specify other attributes of the message, such as carbon copies, blind carbon copies, etc:

```
Mail::send('emails.welcome', $data, function ($message) {
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');
});
```

Here is a list of the available methods on the `$message` message builder instance:

```
$message->from($address, $name = null);
$message->sender($address, $name = null);
$message->to($address, $name = null);
$message->cc($address, $name = null);
$message->bcc($address, $name = null);
$message->replyTo($address, $name = null);
$message->subject($subject);
$message->priority($level);
$message->attach($pathToFile, array $options = []);

// Attach a file from a raw $data string...
$message->attachData($data, $name, array $options = []);
```

```
// Get the underlying SwiftMailer message instance...
$message->getSwiftMessage();
```

> **Note:** The message instance passed to a `Mail::send` Closure extends the SwiftMailer message class, allowing you to call any method on that class to build your e-mail messages.

Mailing Plain Text

By default, the view given to the `send` method is assumed to contain HTML. However, by passing an array as the first argument to the `send` method, you may specify a plain text view to send in addition to the HTML view:

```
Mail::send(['html.view', 'text.view'], $data, $callback);
```

Or, if you only need to send a plain text e-mail, you may specify this using the `text` key in the array:

```
Mail::send(['text' => 'view'], $data, $callback);
```

Mailing Raw Strings

You may use the `raw` method if you wish to e-mail a raw string directly:

```
Mail::raw('Text to e-mail', function ($message) {
    //
});
```

Attachments

To add attachments to an e-mail, use the `attach` method on the `$message` object passed to your Closure. The `attach` method accepts the full path to the file as its first argument:

```
Mail::send('emails.welcome', $data, function ($message) {
    //

    $message->attach($pathToFile);
});
```

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:

```
$message->attach($pathToFile, ['as' => $display, 'mime' => $mime]);
```

Inline Attachments

Embedding An Image In An E-Mail View

Embedding inline images into your e-mails is typically cumbersome; however, Laravel provides a convenient way to attach images to your e-mails and retrieving the appropriate CID. To embed an inline image, use the `embed` method on the `$message` variable within your e-mail view. Remember, Laravel automatically makes the `$message` variable available to all of your e-mail views:

```
<body>
    Here is an image:

    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

Embedding Raw Data In An E-Mail View

If you already have a raw data string you wish to embed into an e-mail message, you may use the `embedData` method on the `$message` variable:

```
<body>
    Here is an image from raw data:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

Queueing Mail

Queueing A Mail Message

Since sending e-mail messages can drastically lengthen the response time of your application, many developers choose to queue e-mail messages for background sending. Laravel makes this easy using its built-in unified queue API. To queue a mail message, use the `queue` method on the `Mail` facade:

```
Mail::queue('emails.welcome', $data, function ($message) {
    //
});
```

This method will automatically take care of pushing a job onto the queue to send the mail message in the background. Of course, you will need to configure your queues before using this feature.

Delayed Message Queueing

If you wish to delay the delivery of a queued e-mail message, you may use the `later` method. To get started, simply pass the number of seconds by which you wish to delay the sending of the message as the first argument to the method:

```
Mail::later(5, 'emails.welcome', $data, function ($message) {
    //
});
```

Pushing To Specific Queues

If you wish to specify a specific queue on which to push the message, you may do so using the `queueOn` and `laterOn` methods:

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function ($message) {
    //
});
```

```
Mail::laterOn('queue-name', 5, 'emails.welcome', $data, function ($message) {
    //
});
```

Mail & Local Development

When developing an application that sends e-mail, you probably don't want to actually send e-mails to live e-mail addresses. Laravel provides several ways to "disable" the actual sending of e-mail messages.

Log Driver

One solution is to use the `log` mail driver during local development. This driver will write all e-mail messages to your log files for inspection. For more information on configuring your application per environment, check out the configuration documentation.

Universal To

Another solution provided by Laravel is to set a universal recipient of all e-mails sent by the framework. This way, all the emails generated by your application will be sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `to` option in your `config/mail.php` configuration file:

```
'to' => [
    'address' => 'dev@domain.com',
    'name' => 'Dev Example'
],
```

Mailtrap

Finally, you may use a service like Mailtrap and the `smtp` driver to send your e-mail messages to a "dummy" mailbox where you may view them in a true e-mail client. This approach has the benefit of allowing you to actually inspect the final e-mails in Mailtrap's message viewer.

Events

Laravel fires the `mailer.sending` event just before sending mail messages. Remember, this event is fired when the mail is *sent*, not when it is queued. You may register an event listener in your `EventServiceProvider`:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    $events->listen('mailer.sending', function ($message) {
        //
    });
}
```

Services

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 an entire BDD testing framework like Behat.

Of course, there are different types of packages. Some packages are stand-alone, meaning they work with any framework, not just Laravel. Both Carbon and Behat are examples of stand-alone packages. Any of these packages may be used with Laravel by simply requesting them in your `composer.json` file.

On the other hand, other packages are specifically intended for use with Laravel. These packages may have routes, controllers, views, and configuration specifically intended to enhance a Laravel application. This guide primarily covers the development of those packages that are Laravel specific.

Service Providers

Service providers are the connection points between your package and Laravel. A service provider is responsible for binding things into Laravel's service container and informing Laravel where to load package resources such as views, configuration, and localization files.

A service provider extends the `Illuminate\Support\ServiceProvider` class and contains two methods: `register` and `boot`. The base `ServiceProvider` class is located in the `illuminate/support` Composer package, which you should add to your own package's dependencies.

To learn more about the structure and purpose of service providers, check out their documentation.

Routing

To define routes for your package, simply `require` the routes file from within your package service provider's `boot` method. From within your routes file, you may use the `Route` facade to register routes just as you would within a typical Laravel application:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    if (! $this->app->routesAreCached()) {
        require __DIR__.'/../../routes.php';
    }
}
```

Resources

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 name is "courier", add the following to your service provider's `boot` method:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

Package views are referenced using a double-colon `package::view` syntax. So, you may load the `admin` view from the `courier` package like so:

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

Overriding Package Views

When you use the `loadViewsFrom` method, Laravel actually registers **two** locations for your views: one in the application's `resources/views/vendor` directory and one in the directory you specify. So, using our `courier` example: when requesting a package view, Laravel will first check if a custom version of the view has been provided by the developer in `resources/views/vendor/courier`. Then, if the view has not been customized, Laravel will search the package view directory you specified in your call to `loadViewsFrom`. This makes it easy for end-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 corresponding publish locations.

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/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 location.

Translations

If your package contains translation files, you may use the `loadTranslationsFrom` method to inform Laravel how to load them. For example, if your package is named "courier", you should add the following to your service provider's `boot` method:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

Package translations are referenced using a double-colon `package::file.line` syntax. 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 corresponding publish locations. For example, to the publish the translation files for our example `courier` package:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => base_path('resources/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 location.

Configuration

Typically, you will want to publish your package's configuration file to the application's own `config` directory. This will allow users of your package to easily override your default configuration options. To publish a configuration file, just use the `publishes` method from the `boot` method of your service provider:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` command, your file will be copied to the specified location. Of course, once your configuration has been published, it can be accessed like any other configuration file:

```
$value = config('courier.option');
```

Default Package Configuration

You may also choose to merge your own package configuration file with the application's copy. This allows your users to include only the options they actually want to override in the published copy of the configuration. To merge the configurations, use the `mergeConfigFrom` method within your service provider's `register` method:

```
/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__.'/path/to/config/courier.php', 'courier'
    );
}
```

Public Assets

Your packages may have assets such as JavaScript, CSS, and images. To publish these assets to the application's `public` directory, use the service provider's `publishes` method. In this example, we will also add a `public` asset group tag, which may be used to publish groups of related assets:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/assets' => public_path('vendor/courier'),
    ], 'public');
}
```

Now, when your package's users execute the `vendor:publish` command, your assets will be copied to the specified location. Since you typically will need to overwrite the assets every time the package is updated, you may use the `--force` flag:

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

If you would like to make sure your public assets are always up-to-date, you can add this command to the `post-update-cmd` list in your `composer.json` file.

Publishing File Groups

You may want to publish groups of package assets and resources separately. For instance, you might want your users to be able to publish your package's configuration files without being forced to publish your package's assets at the same time. You may do this by "tagging" them when calling the `publishes` method. For example, let's define two publish groups in the `boot` method of a package service provider:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'migrations');
}
```

Now your users may publish these groups separately by referencing their tag name when using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --provider="Vendor\Providers\PackageServiceProvider" --tag="config"
```

Services

Pagination

Introduction

In other frameworks, pagination can be very painful. Laravel makes it a breeze. Laravel can quickly generate an intelligent "range" of links based on the current page, and the generated HTML is compatible with the Bootstrap CSS framework.

Basic Usage

Paginating Query Builder Results

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent query. The `paginate` method provided by Laravel automatically takes care of setting the proper limit and offset based on the current page being viewed by the user. By default, the current page is detected by the value of the `?page` query string argument on the HTTP request. Of course, this value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

First, let's take a look at calling the `paginate` method on a query. In this example, the only argument passed to `paginate` is the number of items you would like displayed "per page". In this case, let's specify that we would like to display `15` items per page:

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show all of the users for the application.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);

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

> **Note:** Currently, pagination operations that use a `groupBy` statement cannot be executed efficiently by Laravel. If you need to use a `groupBy` with a paginated result set, it is recommended that you query the database and create a paginator manually.

"Simple Pagination"

If you only need to display simple "Next" and "Previous" links in your pagination view, you have the option of using the `simplePaginate` method to perform a more efficient query. This is very useful for large datasets if you do not need to display a link for each page number when rendering your view:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries. In this example, we will paginate the `User` model with `15` items per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
$users = App\User::paginate(15);
```

Of course, you may call `paginate` after setting other constraints on the query, such as `where` clauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the `simplePaginate` method when paginating Eloquent models:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Manually Creating A Paginator

Sometimes you may wish to create a pagination instance manually, passing it an array of items. You may do so by creating either an `Illuminate\Pagination\Paginator` or `Illuminate\Pagination\LengthAwarePaginator` instance, depending on your needs.

The `Paginator` class does not need to know the total number of items in the result set; however, because of this, the class does not have methods for retrieving the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`; however, it does require a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simplePaginate` method on the query builder and Eloquent, while the `LengthAwarePaginator` corresponds to the `paginate` method.

When manually creating a paginator instance, you should manually "slice" the array of results you pass to the paginator. If you're unsure how to do this, check out the array_slice PHP function.

Displaying Results In A View

When you call the `paginate` or `simplePaginate` methods on a query builder or Eloquent query, you will receive a paginator instance. When calling the `paginate` method, you will receive an instance of `Illuminate\Pagination\LengthAwarePaginator`. When calling the `simplePaginate` method, you will receive an instance of `Illuminate\Pagination\Paginator`. These objects provide several methods that describe the result set. In addition to these helpers methods, the paginator instances are iterators and may be looped as an array.

So, once you have retrieved the results, you may display the results and render the page links using Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{!! $users->render() !!}
```

The `render` method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper `?page` query string variable. Remember, the HTML generated by the `render` method is compatible with the Bootstrap CSS framework.

> **Note:** When calling the `render` method from a Blade template, be sure to use the `{!! !!}` syntax so the HTML links are not escaped.

Customizing The Paginator URI

The `setPath` method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like `http://example.com/custom/url?page=N`, you should

pass `custom/url` to the `setPath` method:

```
Route::get('users', function () {
    $users = App\User::paginate(15);

    $users->setPath('custom/url');

    //
});
```

Appending To Pagination Links

You may add to the query string of pagination links using the `appends` method. For example, to append `&sort=votes` to each pagination link, you should make the following call to `appends`:

```
{!! $users->appends(['sort' => 'votes'])->render() !!}
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the `fragment` method. For example, to append `#foo` to the end of each pagination link, make the following call to the `fragment` method:

```
{!! $users->fragment('foo')->render() !!}
```

Additional Helper Methods

You may also access additional pagination information via the following methods on paginator instances:

- `$results->count()`
- `$results->currentPage()`
- `$results->hasMorePages()`
- `$results->lastPage()` (Not available when using `simplePaginate`)
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total()` (Not available when using `simplePaginate`)
- `$results->url($page)`

Converting Results To JSON

The Laravel paginator result classes implement the `Illuminate\Contracts\Support\JsonableInterface` contract and expose the `toJson` method, so it's very easy to convert your pagination results to JSON.

You may also convert a paginator instance to JSON by simply returning it from a route or controller action:

```
Route::get('users', function () {
    return App\User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The actual result objects will be available via the `data` key in the JSON array. Here is an example of the JSON created by returning a paginator instance from a route:

Example Paginator JSON

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "from": 1,
   "to": 15,
   "data":[
        {
            // Result Object
        },
        {
```

```
            // Result Object
        }
    ]
}
```

Services

Queues

Introduction

The Laravel queue service provides a unified API across a variety of different queue back-ends. Queues allow you to defer the processing of a time consuming task, such as sending an e-mail, until a later time which drastically speeds up web requests to your application.

Configuration

The queue configuration file is stored in `config/queue.php`. In this file you will find connection configurations for each of the queue drivers that are included with the framework, which includes a database, Beanstalkd, IronMQ, Amazon SQS, Redis, and synchronous (for local use) driver.

A `null` queue driver is also included which simply discards queued jobs.

Driver Prerequisites

Database

In order to use the `database` queue driver, you will need a database table to hold the jobs. To generate a migration that creates this table, run the `queue:table` Artisan command. Once the migration is created, you may migrate your database using the `migrate` command:

```
php artisan queue:table

php artisan migrate
```

Other Queue Dependencies

The following dependencies are needed for the listed queue drivers:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq ~2.0|~4.0`
- Redis: `predis/predis ~1.0`

Writing Job Classes

Generating Job Classes

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. You may generate a new queued job using the Artisan CLI:

```
php artisan make:job SendReminderEmail --queued
```

This command will generate a new class in the `app/Jobs` directory, and the class will implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating to Laravel that the job should be pushed onto the queue instead of run synchronously.

Job Class Structure

Job classes are very simple, normally containing only a `handle` method which is called when the job is processed by the queue. To get started, let's take a look at an example job class:

```
<?php

namespace App\Jobs;

use App\User;
use App\Jobs\Job;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, ShouldQueue
{
    use InteractsWithQueue, SerializesModels;

    protected $user;

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

    /**
     * Execute the job.
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        $mailer->send('emails.reminder', ['user' => $this->user], function ($m) {
            //
        });

        $this->user->reminders()->create(...);
    }
}
```

In this example, note that we were able to pass an Eloquent model directly into the queued job's constructor. Because of the `SerializesModels` trait that the job is using, Eloquent models will be gracefully serialized and unserialized when the job is processing. If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance from the database. It's all totally transparent to your application and prevents issues that can arise from serializing full Eloquent model instances.

The `handle` method is called when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Laravel service container automatically injects these dependencies.

When Things Go Wrong

If an exception is thrown while the job is being processed, it 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 number of maximum attempts is defined by the `--tries` switch used on the `queue:listen` or `queue:work` Artisan jobs. More information on running the queue listener can be found below.

Manually Releasing Jobs

If you would like to `release` the job manually, the `InteractsWithQueue` trait, which is already included in your generated job class, provides access to the queue job `release` method. The `release` method accepts one argument: the number of seconds you wish to wait until the job is made available again:

```
public function handle(Mailer $mailer)
{
    if (condition) {
        $this->release(10);
    }
}
```

Checking The Number Of Run Attempts

As noted above, if an exception occurs while the job is being processed, it will automatically be released back onto the queue. You may check the number of attempts that have been made to run the job using the `attempts` method:

```
public function handle(Mailer $mailer)
{
    if ($this->attempts() > 3) {
        //
    }
}
```

Pushing Jobs Onto The Queue

The default Laravel controller located in `app/Http/Controllers/Controller.php` uses a `DispatchesJobs` trait. This trait provides several methods allowing you to conveniently push jobs onto the queue, such as the `dispatch` method:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Send a reminder e-mail to a given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $this->dispatch(new SendReminderEmail($user));
    }
}
```

Of course, sometimes you may wish to dispatch a job from somewhere in your application besides a route or controller. For that reason, you can include the `DispatchesJobs` trait on any of the classes in your application to gain access to its various dispatch methods. For example, here is a sample class that uses the trait:

```
<?php
```

```
namespace App;

use Illuminate\Foundation\Bus\DispatchesJobs;

class ExampleClass
{
    use DispatchesJobs;
}
```

Specifying The Queue For A Job

You may also specify the queue a job should be sent to.

By pushing jobs to different queues, you may "categorize" your queued jobs, and even prioritize how many workers you assign to various queues. This does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method on the job instance. The `onQueue` method is provided by the `Illuminate\Bus\Queueable` trait, which is already included on the `App\Jobs\Job` base class:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Send a reminder e-mail to a given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $job = (new SendReminderEmail($user))->onQueue('emails');

        $this->dispatch($job);
    }
}
```

Delayed Jobs

Sometimes you may wish to delay the execution of a queued job. For instance, you may wish to queue a job that sends a customer a reminder e-mail 15 minutes after sign-up. You may accomplish this using the `delay` method on your job class, which is provided by the `Illuminate\Bus\Queueable` trait:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Send a reminder e-mail to a given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);
```

```
        $job = (new SendReminderEmail($user))->delay(60);

        $this->dispatch($job);
    }
}
```

In this example, we're specifying that the job should be delayed in the queue for 60 seconds before being made available to workers.

Note: The Amazon SQS service has a maximum delay time of 15 minutes.

Dispatching Jobs From Requests

It is very common to map HTTP request variables into jobs. So, instead of forcing you to do this manually for each request, Laravel provides some helper methods to make it a cinch. Let's take a look at the `dispatchFrom` method available on the `DispatchesJobs` trait. By default, this trait is included on the base Laravel controller class:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class CommerceController extends Controller
{
    /**
     * Process the given order.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function processOrder(Request $request, $id)
    {
        // Process the request...

        $this->dispatchFrom('App\Jobs\ProcessOrder', $request);
    }
}
```

This method will examine the constructor of the given job class and extract variables from the HTTP request (or any other `ArrayAccess` object) to fill the needed constructor parameters of the job. So, if our job class accepts a `productId` variable in its constructor, the job bus will attempt to pull the `productId` parameter from the HTTP request.

You may also pass an array as the third argument to the `dispatchFrom` method. This array will be used to fill any constructor parameters that are not available on the request:

```
$this->dispatchFrom('App\Jobs\ProcessOrder', $request, [
    'taxPercentage' => 20,
]);
```

Job Events

Job Completion Event

The `Queue::after` method allows you to register a callback to be executed when a queued job executes successfully. This callback is a great opportunity to perform additional logging, queue a subsequent job, or increment statistics for a dashboard. For example, we may attach a callback to this event from the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;
```

```
class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::after(function ($connection, $job, $data) {
            //
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Running The Queue Listener

Starting The Queue Listener

Laravel includes an Artisan command that will run new jobs as they are pushed onto the queue. You may run the listener using the `queue:listen` command:

```
php artisan queue:listen
```

You may also specify which queue connection the listener should utilize:

```
php artisan queue:listen connection
```

Note that once this task has started, it will continue to run until it is manually stopped. You may use a process monitor such as Supervisor to ensure that the queue listener does not stop running.

Queue Priorities

You may pass a comma-delimited list of queue connections to the `listen` job to set queue priorities:

```
php artisan queue:listen --queue=high,low
```

In this example, jobs on the `high` queue will always be processed before moving onto jobs from the `low` queue.

Specifying The Job Timeout Parameter

You may also set the length of time (in seconds) each job should be allowed to run:

```
php artisan queue:listen --timeout=60
```

Specifying Queue Sleep Duration

In addition, you may specify the number of seconds to wait before polling for new jobs:

```
php artisan queue:listen --sleep=5
```

Note that the queue only "sleeps" if no jobs are on the queue. If more jobs are available, the queue will continue to work them without sleeping.

Supervisor Configuration

Supervisor is a process monitor for the Linux operating system, and will automatically restart your `queue:listen` or `queue:work` commands if they fail. To install Supervisor on Ubuntu, you may use the following command:

```
sudo apt-get install supervisor
```

Supervisor configuration files are typically stored in the `/etc/supervisor/conf.d` directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a `laravel-worker.conf` file that starts and monitors a `queue:work` process:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --daemon
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

In this example, the `numprocs` directive will instruct Supervisor to run 8 `queue:work` processes and monitor all of them, automatically restarting them if they fail. Of course, you should change the `queue:work sqs` portion of the `command` directive to reflect your chosen queue connection.

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 configuring and using Supervisor, consult the Supervisor documentation. Alternatively, you may use Laravel Forge to automatically configure and manage your Supervisor configuration from a convenient web interface.

Daemon Queue Listener

The `queue:work` Artisan command includes a `--daemon` option for forcing the queue worker to continue processing jobs without ever re-booting the framework. This results in a significant reduction of CPU usage when compared to the `queue:listen` command:

To start a queue worker in daemon mode, use the `--daemon` flag:

```
php artisan queue:work connection --daemon

php artisan queue:work connection --daemon --sleep=3

php artisan queue:work connection --daemon --sleep=3 --tries=3
```

As you can see, the `queue:work` job supports most of the same options available to `queue:listen`. You may use the `php artisan help queue:work` job to view all of the available options.

Coding Considerations For Daemon Queue Listeners

Daemon queue workers do not restart the framework before processing each job. Therefore, you should be careful to free any heavy resources before your job finishes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done.

Similarly, your database connection may disconnect when being used by a long-running daemon. You may use the `DB::reconnect` method to ensure you have a fresh connection.

Deploying With Daemon Queue Listeners

Since daemon queue workers are long-lived processes, they will not pick up changes in your code without

being restarted. So, the simplest way to deploy an application using daemon queue workers is to restart the workers during your deployment script. You may gracefully restart all of the workers by including the following command in your deployment script:

```
php artisan queue:restart
```

This command will gracefully instruct all queue workers to restart after they finish processing their current job so that no existing jobs are lost.

> **Note:** This command relies on the cache system to schedule the restart. By default, APCu does not work for CLI jobs. If you are using APCu, add `apc.enable_cli=1` to your APCu configuration.

Dealing With Failed Jobs

Since things don't always go as planned, sometimes your queued jobs will fail. Don't worry, it happens to the best of us! Laravel includes a convenient way to specify the maximum number of times a job should be attempted. After a job has exceeded this amount of attempts, it will be inserted into a `failed_jobs` table. The name of the table can be configured via the `config/queue.php` configuration file.

To create a migration for the `failed_jobs` table, you may use the `queue:failed-table` command:

```
php artisan queue:failed-table
```

When running your queue listener, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:listen` command:

```
php artisan queue:listen connection-name --tries=3
```

Failed Job Events

If you would like to register an event that will be called when a queued job fails, you may use the `Queue::failing` method. This event is a great opportunity to notify your team via e-mail or HipChat. For example, we may attach a callback to this event from the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function ($connection, $job, $data) {
            // Notify team of failing job...
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Failed Method On Job Classes

For more granular control, you may define a `failed` method directly on a queue job class, allowing you to

perform job specific actions when a failure occurs:

```
<?php

namespace App\Jobs;

use App\Jobs\Job;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, ShouldQueue
{
    use InteractsWithQueue, SerializesModels;

    /**
     * Execute the job.
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        //
    }

    /**
     * Handle a job failure.
     *
     * @return void
     */
    public function failed()
    {
        // Called when the job is failing...
    }
}
```

Retrying Failed Jobs

To view all of your failed jobs that have been inserted into your `failed_jobs` database table, you may use the `queue:failed` Artisan command:

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, and failure time. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of 5, the following command should be issued:

```
php artisan queue:retry 5
```

To retry all of your failed jobs, use `queue:retry` with `all` as the ID:

```
php artisan queue:retry all
```

If you would like to delete a failed job, you may use the `queue:forget` command:

```
php artisan queue:forget 5
```

To delete all of your failed jobs, you may use the `queue:flush` command:

```
php artisan queue:flush
```

Services

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, you will need to install the `predis/predis` package (~1.0) via Composer.

Configuration

The Redis configuration for your application is located in the `config/database.php` configuration file. Within this file, you will see a `redis` array containing the Redis servers used by your application:

```
'redis' => [

    'cluster' => false,

    'default' => [
        'host'     => '127.0.0.1',
        'port'     => 6379,
        'database' => 0,
    ],

],
```

The default server configuration should suffice for development. However, you are free to modify this array based on your environment. Simply give each Redis server a name, and specify the host and port used by the server.

The `cluster` option will tell the Laravel Redis client to perform client-side sharding across your Redis nodes, allowing you to pool nodes and create a large amount of available RAM. However, note that client-side sharding does not handle failover; therefore, is primarily suited for cached data that is available from another primary data store.

Additionally, you may define an `options` array value in your Redis connection definition, allowing you to specify a set of Predis client options.

If your Redis server requires authentication, you may supply a password by adding a `password` configuration item to your Redis server configuration array.

> **Note:** If you have the Redis PHP extension installed via PECL, you will need to rename the alias for Redis in your `config/app.php` file.

Basic Usage

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 `GET` command on Redis by calling the `get` method on the `Redis` facade:

```
<?php

namespace App\Http\Controllers;

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

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

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

Of course, as mentioned above, you may call any of the Redis commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server, so simply pass the arguments the Redis command expects:

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may also pass commands to the server using the `command` method, which accepts the name of the command as its first argument, and an array of values as its second argument:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Using Multiple Redis Connections

You may get a Redis instance by calling the `Redis::connection` method:

```
$redis = Redis::connection();
```

This will give you an instance of the default Redis server. If you are not using server clustering, you may pass the server name to the `connection` method to get a specific server as defined in your Redis configuration:

```
$redis = Redis::connection('other');
```

Pipelining Commands

Pipelining should be used when you need to send many commands to the server in one operation. The `pipeline` method accepts one argument: a `Closure` that receives a Redis instance. You may issue all of your commands to this Redis instance and they will all be executed within a single operation:

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel also 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 / processes.

First, let's setup a listener on a channel via Redis using the `subscribe` method. We will place this method call within an Artisan command since calling the `subscribe` method begins a long-running process:

```
<?php

namespace App\Console\Commands;

use Redis;
use Illuminate\Console\Command;
```

```
class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function($message) {
            echo $message;
        });
    }
}
```

Now, we may publish messages to the channel using the `publish` method:

```
Route::get('publish', function () {
    // Route logic...

    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

Wildcard Subscriptions

Using the `psubscribe` method, you may subscribe to a wildcard channel, which is useful for catching all messages on all channels. The `$channel` name will be passed as the second argument to the provided callback `Closure`:

```
Redis::psubscribe(['*'], function($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function($message, $channel) {
    echo $message;
});
```

Services

Session

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across requests. Laravel ships with a variety of session back-ends available for use through a clean, unified API. Support for popular back-ends such as Memcached, Redis, and databases is included out of the box.

Configuration

The session configuration file is stored at `config/session.php`. Be sure to review the well documented options available to you in this file. By default, Laravel is configured to use the `file` session driver, which will work well for many applications. In production applications, you may consider using the `memcached` or `redis` drivers for even faster session performance.

The session `driver` defines where session data will be stored for each request. Laravel ships with several great drivers out of the box:

- `file` - sessions are stored in `storage/framework/sessions`.
- `cookie` - sessions are stored in secure, encrypted cookies.
- `database` - sessions are stored in a database used by your application.
- `memcached` / `redis` - sessions are stored in one of these fast, cache based stores.
- `array` - sessions are stored in a simple PHP array and will not be persisted across requests.

Note: The array driver is typically used for running tests to prevent session data from persisting.

Driver Prerequisites

Database

When using the `database` session driver, you will need to setup a table to contain the session items. Below is an example `Schema` declaration for the table:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

You may use the `session:table` Artisan command to generate this migration for you!

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer.

Other Session Considerations

The Laravel framework uses the `flash` session key internally, so you should not add an item to the session by that name.

If you need all stored session data to be encrypted, set the `encrypt` configuration option to `true`.

Basic Usage

Accessing The Session

First, let's access the session. We can access the session instance via the HTTP request, which can be type-hinted on a controller method. Remember, controller method dependencies are injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function showProfile(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

When you retrieve a value from the session, you may also pass a default value as the second argument to the `get` method. This default value will be returned if the specified key does not exist in the session. If you pass a `Closure` as the default value to the `get` method, the `Closure` will be executed and its result returned:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function() {
    return 'default';
});
```

If you would like to retrieve all data from the session, you may use the `all` method:

```
$data = $request->session()->all();
```

You may also use the global `session` PHP function to retrieve and store data in the session:

```
Route::get('home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

Determining If An Item Exists In The Session

The `has` method may be used to check if an item exists in the session. This method will return `true` if the item exists:

```
if ($request->session()->has('users')) {
    //
}
```

Storing Data In The Session

Once you have access to the session instance, you may call a variety of functions to interact with the underlying data. For example, the `put` method stores a new piece of data in the session:

```
$request->session()->put('key', 'value');
```

Pushing To Array Session Values

The `push` method may be used to push a new value onto a session value that is an array. For example, if the `user.teams` key contains an array of team names, you may push a new value onto the array like so:

```
$request->session()->push('user.teams', 'developers');
```

Retrieving And Deleting An Item

The `pull` method will retrieve and delete an item from the session:

```
$value = $request->session()->pull('key', 'default');
```

Deleting Items From The Session

The `forget` method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the `flush` method:

```
$request->session()->forget('key');

$request->session()->flush();
```

Regenerating The Session ID

If you need to regenerate the session ID, you may use the `regenerate` method:

```
$request->session()->regenerate();
```

Flash Data

Sometimes you may wish to store items in the session only for the next request. You may do so using the `flash` method. Data stored in the session using this method will only be available during the subsequent HTTP request, and then will be deleted. Flash data is primarily useful for short-lived status messages:

```
$request->session()->flash('status', 'Task was successful!');
```

If you need to keep your flash data around for even more requests, you may use the `reflash` method, which will keep all of the flash data around for an additional request. If you only need to keep specific flash data around, you may use the `keep` method:

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

Adding Custom Session Drivers

To add additional drivers to Laravel's session back-end, you may use the `extend` method on the `Session` facade. You can call the `extend` method from the `boot` method of a service provider:

```
<?php

namespace App\Providers;

use Session;
use App\Extensions\MongoSessionStore;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
```

```
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore;
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Note that your custom session driver should implement the `SessionHandlerInterface`. This interface contains just a few simple methods we need to implement. A stubbed MongoDB implementation looks something like this:

```
<?php

namespace App\Extensions;

class MongoHandler implements SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

Since these methods are not as readily understandable as the cache `StoreInterface`, let's quickly cover what each of the methods do:

- The `open` method would typically be used in file based session store systems. Since Laravel ships with a `file` session driver, you will almost never need to put anything in this method. You can leave it as an empty stub. It is simply a fact of poor interface design (which we'll discuss later) that PHP requires us to implement this method.
- The `close` method, like the `open` method, can also usually be disregarded. For most drivers, it is not needed.
- The `read` method should return the string version of the session data associated with the given `$sessionId`. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
- The `write` method should write the given `$data` string associated with the `$sessionId` to some persistent storage system, such as MongoDB, Dynamo, etc.
- The `destroy` method should remove the data associated with the `$sessionId` from persistent storage.
- The `gc` method should destroy all session data that is older than the given `$lifetime`, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

Once the session driver has been registered, you may use the `mongo` driver in your `config/session.php` configuration file.

Services

Envoy Task Runner

Introduction

Laravel Envoy provides a clean, minimal syntax for defining common tasks you run on your remote servers. Using a Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems.

Installation

First, install Envoy using the Composer `global` command:

```
composer global require "laravel/envoy=~1.0"
```

Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `envoy` executable is found when you run the `envoy` command in your terminal.

Updating Envoy

You may also use Composer to keep your Envoy installation up to date:

```
composer global update
```

Writing Tasks

All of your Envoy tasks should be defined in an `Envoy.blade.php` file in the root of your project. Here's an example to get you started:

```
@servers(['web' => 'user@192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

As you can see, an array of `@servers` is defined at the top of the file, allowing you to reference these servers in the `on` option of your task declarations. Within your `@task` declarations, you should place the Bash code that will be run on your server when the task is executed.

Bootstrapping

Sometimes, you may need to execute some PHP code before evaluating your Envoy tasks. You may use the `@setup` directive to declare variables and do general PHP work inside the Envoy file:

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@endsetup
```

You may also use `@include` to include any outside PHP files:

```
@include('vendor/autoload.php')
```

Confirming Tasks

If you would like to be prompted for confirmation before running a given task on your servers, you may add the `confirm` directive to your task declaration:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Task Variables

If needed, you may pass variables into the Envoy file using command line switches, allowing you to customize your tasks:

```
envoy run deploy --branch=master
```

You may use the options in your tasks via Blade's "echo" syntax:

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Multiple Servers

You may 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, simply list the servers in the task declaration's `on` array:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

By default, the task will be executed on each server serially. Meaning, the task will finish running on the first server before proceeding to execute on the next server.

Parallel Execution

If you would like to run a task across multiple servers in parallel, add the `parallel` option to your task declaration:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Task Macros

Macros allow you to define a set of tasks to be run in sequence using a single command. For instance, a `deploy` macro may run the `git` and `composer` tasks:

```
@servers(['web' => '192.168.1.1'])

@macro('deploy')
```

```
    git
    composer
@endmacro

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask
```

Once the macro has been defined, you may run it via single, simple command:

```
envoy run deploy
```

Running Tasks

To run a task from your `Envoy.blade.php` file, execute Envoy's `run` command, passing the command the name of the task or macro you would like to execute. Envoy will run the task and display the output from the servers as the task is running:

```
envoy run task
```

Notifications

HipChat

After running a task, you may send a notification to your team's HipChat room using Envoy's `@hipchat` directive. The directive accepts an API token, the name of the room, and the username to be displayed as the sender of the message:

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask

@after
    @hipchat('token', 'room', 'Envoy')
@endafter
```

If you wish, you may also pass a custom message to send to the HipChat room. Any variables available to your Envoy tasks will also be available when constructing the message:

```
@after
    @hipchat('token', 'room', 'Envoy', "{{ $task }} ran in the {{ $env }} environment.")
@endafter
```

Slack

In addition to HipChat, Envoy also supports sending notifications to Slack. The `@slack` directive accepts a Slack hook URL, a channel name, and the message you wish to send to the channel:

```
@after
    @slack('hook', 'channel', 'message')
@endafter
```

You may retrieve your webhook URL by creating an `Incoming WebHooks` integration on Slack's website. The `hook` argument should be the entire webhook URL provided by the Incoming Webhooks Slack Integration. For example:

```
https://hooks.slack.com/services/ZZZZZZZZZ/YYYYYYYYY/XXXXXXXXXXXXXXX
```

You may provide one of the following as the channel argument:

- To send the notification to a channel: `#channel`

- To send the notification to a user: `@user`

Services

Task Scheduling

Introduction

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

Your task schedule is defined in the `app/Console/Kernel.php` file's `schedule` method. To help you get started, a simple example is included with the method. You are free to add as many scheduled tasks as you wish to the `Schedule` object.

Starting The Scheduler

Here is the only Cron entry you need to add to your server:

```
* * * * * php /path/to/artisan schedule:run >> /dev/null 2>&1
```

This Cron will call the Laravel command scheduler every minute. Then, Laravel evaluates your scheduled tasks and runs the tasks that are due.

Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of the `App\Console\Kernel` class. To get started, let's look at an example of scheduling a task. In this example, we will schedule a `Closure` to be called every day at midnight. Within the `Closure` we will execute a database query to clear a table:

```
<?php

namespace App\Console;

use DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        \App\Console\Commands\Inspire::class,
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
```

```
        })->daily();
    }
}
```

In addition to scheduling `Closure` calls, you may also schedule Artisan commands and operating system commands. For example, you may use the `command` method to schedule an Artisan command:

```
$schedule->command('emails:send --force')->daily();
```

The `exec` command may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

Of course, there are a variety of schedules you may assign to your task:

Method	Description
`->cron('* * * * * *');`	Run the task on a custom Cron schedule
`->everyMinute();`	Run the task every minute
`->everyFiveMinutes();`	Run the task every five minutes
`->everyTenMinutes();`	Run the task every ten minutes
`->everyThirtyMinutes();`	Run the task every thirty minutes
`->hourly();`	Run the task every hour
`->daily();`	Run the task every day at midnight
`->dailyAt('13:00');`	Run the task every day at 13:00
`->twiceDaily(1, 13);`	Run the task daily at 1:00 & 13:00
`->weekly();`	Run the task every week
`->monthly();`	Run the task every month
`->yearly();`	Run the task every year

These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, to schedule a command to run weekly on Monday:

```
$schedule->call(function () {
    // Runs once a week on Monday at 13:00...
})->weekly()->mondays()->at('13:00');
```

Below is a list of the additional schedule constraints:

Method	Description
`->weekdays();`	Limit the task to weekdays
`->sundays();`	Limit the task to Sunday
`->mondays();`	Limit the task to Monday
`->tuesdays();`	Limit the task to Tuesday
`->wednesdays();`	Limit the task to Wednesday
`->thursdays();`	Limit the task to Thursday
`->fridays();`	Limit the task to Friday
`->saturdays();`	Limit the task to Saturday
`->when(Closure);`	Limit the task based on a truth test

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;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return `true`.

Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` Artisan command will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
         ->daily()
         ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
         ->daily()
         ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may e-mail the output to an e-mail address of your choice. Note that the output must first be sent to a file using the `sendOutputTo` method. Also, before e-mailing the output of a task, you should configure Laravel's e-mail services:

```
$schedule->command('foo')
         ->daily()
         ->sendOutputTo($filePath)
         ->emailOutputTo('foo@example.com');
```

> **Note:** The `emailOutputTo` and `sendOutputTo` methods are exclusive to the `command` method and are not supported for `call`.

Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is complete:

```
$schedule->command('emails:send')
         ->daily()
         ->before(function () {
             // Task is about to start...
         })
         ->after(function () {
             // Task is complete...
         });
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is complete. This method is useful for notifying an external service, such as Laravel Envoyer, that your scheduled task is commencing or complete:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBefore($url)
         ->thenPing($url);
```

Using either the `pingBefore($url)` or `thenPing($url)` feature requires the Guzzle HTTP library. You can add

Guzzle to your project by adding the following line to your `composer.json` file:

```
"guzzlehttp/guzzle": "~5.3|~6.0"
```

Services

Testing

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box, and a `phpunit.xml` file is already setup for your application. The framework also ships with convenient helper methods allowing you to expressively test your applications.

An `ExampleTest.php` file is provided in the `tests` directory. After installing a new Laravel application, simply run `phpunit` on the command line to run your tests.

Test Environment

When running tests, Laravel will automatically set the configuration environment to `testing`. Laravel 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 create other testing environment configurations as necessary. The `testing` environment variables may be configured in the `phpunit.xml` file.

Defining & Running Tests

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

```
php artisan make:test UserTest
```

This command will place a new `UserTest` class within your `tests` directory. You may then define test methods as you normally would using PHPUnit. To run your tests, simply execute the `phpunit` command from your terminal:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class UserTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
```

```
        $this->assertTrue(true);
    }
}
```

Note: If you define your own `setUp` method within a test class, be sure to call `parent::setUp`.

Application Testing

Laravel provides a very fluent API for making HTTP requests to your application, examining the output, and even filling out forms. For example, take a look at the `ExampleTest.php` file included in your `tests` directory:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5')
             ->dontSee('Rails');
    }
}
```

The `visit` method makes a `GET` request into the application. The `see` method asserts that we should see the given text in the response returned by the application. The `dontSee` method asserts that the given text is not returned in the application response. This is the most basic application test available in Laravel.

Interacting With Your Application

Of course, you can do much more than simply assert that text appears in a given response. Let's take a look at some examples of clicking links and filling out forms:

Clicking Links

In this test, we will make a request to the application, "click" a link in the returned response, and then assert that we landed on a given URI. For example, let's assume there is a link in our response that has a text value of "About Us":

```
<a href="/about-us">About Us</a>
```

Now, let's write a test that clicks the link and asserts the user lands on the correct page:

```
public function testBasicExample()
{
    $this->visit('/')
         ->click('About Us')
         ->seePageIs('/about-us');
}
```

Working With Forms

Laravel also provides several methods for testing forms. The `type`, `select`, `check`, `attach`, and `press` methods allow you to interact with all of your form's inputs. For example, let's imagine this form exists on the application's registration page:

```
<form action="/register" method="POST">
    {!! csrf_field() !!}

    <div>
        Name: <input type="text" name="name">
    </div>
```

```
    <div>
        <input type="checkbox" value="yes" name="terms"> Accept Terms
    </div>

    <div>
        <input type="submit" value="Register">
    </div>
</form>
```

We can write a test to complete this form and inspect the result:

```
public function testNewUserRegistration()
{
    $this->visit('/register')
         ->type('Taylor', 'name')
         ->check('terms')
         ->press('Register')
         ->seePageIs('/dashboard');
}
```

Of course, if your form contains other inputs such as radio buttons or drop-down boxes, you may easily fill out those types of fields as well. Here is a list of each form manipulation method:

Method	Description
`$this->type($text, $elementName)`	"Type" text into a given field.
`$this->select($value, $elementName)`	"Select" a radio button or drop-down field.
`$this->check($elementName)`	"Check" a checkbox field.
`$this->attach($pathToFile, $elementName)`	"Attach" a file to the form.
`$this->press($buttonTextOrElementName)`	"Press" a button with the given text or name.

Working With Attachments

If your form contains `file` input types, you may attach files to the form using the `attach` method:

```
public function testPhotoCanBeUploaded()
{
    $this->visit('/upload')
         ->name('File Name', 'name')
         ->attach($absolutePathToFile, 'photo')
         ->press('Upload')
         ->see('Upload Successful!');
}
```

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `get`, `post`, `put`, `patch`, and `delete` methods may be used to issue requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/user` and assert that a given array was returned in JSON format:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJson([
                 'created' => true,
             ]);
    }
}
```

The `seeJson` method converts the given array into JSON, and then verifies that the JSON fragment occurs

anywhere within the entire 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.

Verify Exact JSON Match

If you would like to verify that the given array is an **exact** match for the JSON returned by the application, you should use the `seeJsonEquals` method:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJsonEquals([
                 'created' => true,
             ]);
    }
}
```

Sessions / Authentication

Laravel provides several helpers for working with the session during 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 testing a request to your application:

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $this->withSession(['foo' => 'bar'])
             ->visit('/');
    }
}
```

Of course, one common use of the session is for maintaining user state, such as the authenticated user. The `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a model factory to generate and authenticate a user:

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(App\User::class)->create();

        $this->actingAs($user)
             ->withSession(['foo' => 'bar'])
             ->visit('/')
             ->see('Hello, '.$user->name);
    }
}
```

Disabling Middleware

When testing your application, you may find it convenient to disable middleware for some of your tests. This will allow you to test your routes and controller in isolation from any middleware concerns. Laravel includes a simple `WithoutMiddleware` trait that you can use to automatically disable all middleware for the test class:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
```

```
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use WithoutMiddleware;

    //
}
```

If you would like to only disable middleware for a few test methods, you may call the `withoutMiddleware` method from within the test methods:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->withoutMiddleware();

        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Custom HTTP Requests

If you would like to make a custom HTTP request into your application and get the full `Illuminate\Http\Response` object, you may use the `call` method:

```
public function testApplication()
{
    $response = $this->call('GET', '/');

    $this->assertEquals(200, $response->status());
}
```

If you are making `POST`, `PUT`, or `PATCH` requests you may pass an array of input data with the request. Of course, this data will be available in your routes and controller via the Request instance:

```
   $response = $this->call('POST', '/user', ['name' => 'Taylor']);
```

PHPUnit Assertions

Laravel provides several additional assertion methods for PHPUnit tests:

Method	Description
`->assertResponseOk();`	Assert that the client response has an OK status code.
`->assertResponseStatus($code);`	Assert that the client response has a given code.
`->assertViewHas($key, $value = null);`	Assert that the response view has a given piece of bound data.
`->assertViewHasAll(array $bindings);`	Assert that the view has a given list of bound data.
`->assertViewMissing($key);`	Assert that the response view is missing a piece of bound data.
`->assertRedirectedTo($uri, $with = []);`	Assert whether the client was redirected to a given URI.
`->assertRedirectedToRoute($name, $parameters = [], $with = []);`	Assert whether the client was redirected to a given route.
`->assertRedirectedToAction($name, $parameters = [], $with = []);`	Assert whether the client was redirected to a given action.
`->assertSessionHas($key, $value = null);`	Assert that the session has a given value.
`->assertSessionHasAll(array $bindings);`	Assert that the session has a given list of values.
`->assertSessionHasErrors($bindings = [], $format = null);`	Assert that the session has errors bound.
`->assertHasOldInput();`	Assert that the session has old input.

Working With Databases

Laravel also provides a variety of helpful tools to make it easier to test your database driven applications. First, you may use the `seeInDatabase` helper to assert that data exists in the database matching a given set of criteria. For example, if we would like to verify that there is a record in the `users` table with the `email` value of `sally@example.com`, we can do the following:

```
public function testDatabase()
{
    // Make call to application...

    $this->seeInDatabase('users', ['email' => 'sally@example.com']);
}
```

Of course, the `seeInDatabase` method and other helpers like it are for convenience. You are free to use any of PHPUnit's built-in assertion methods to supplement your tests.

Resetting The Database After Each Test

It is often useful to reset your database after each test so that data from a previous test does not interfere with subsequent tests.

Using Migrations

One option is to rollback the database after each test and migrate it before the next test. Laravel provides a simple `DatabaseMigrations` trait that will automatically handle this for you. Simply use the trait on your test class:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Using Transactions

Another option is to wrap every test case in a database transaction. Again, Laravel provides a convenient `DatabaseTransactions` trait that will automatically handle this:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseTransactions;

    /**
     * A basic functional test example.
     *
     * @return void
     */
```

```
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Note: This trait will only wrap the default database connection in a transaction.

Model Factories

When testing, it is common to need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Laravel allows you to define a default set of attributes for each of your Eloquent models using "factories". To get started, take a look at the `database/factories/ModelFactory.php` file in your application. Out of the box, this file contains one factory definition:

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

Within the Closure, which serves as the factory definition, you may return the default test values of all attributes on the model. The Closure will receive an instance of the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

Of course, you are free to add your own additional factories to the `ModelFactory.php` file.

Multiple Factory Types

Sometimes you may wish to have multiple factories for the same Eloquent model class. For example, perhaps you would like to have a factory for "Administrator" users in addition to normal users. You may define these factories using the `defineAs` method:

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

Instead of duplicating all of the attributes from your base user factory, you may use the `raw` method to retrieve the base attributes. Once you have the attributes, simply supplement them with any additional values you require:

```
$factory->defineAs(App\User::class, 'admin', function ($faker) use ($factory) {
    $user = $factory->raw(App\User::class);

    return array_merge($user, ['admin' => true]);
});
```

Using Factories In Tests

Once you have defined your factories, you may use them in your tests or database seed files to generate model instances using the global `factory` function. So, let's take a look at a few examples of creating models. First, we'll use the `make` method, which creates models but does not save them to the database:

```
public function testDatabase()
{
    $user = factory(App\User::class)->make();

    // Use model in tests...
```

```
}
```

If you would like to override some of the default values of your models, you may pass an array of values to the `make` method. Only the specified values will be replaced while the rest of the values remain set to their default values as specified by the factory:

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
   ]);
```

You may also create a Collection of many models or create models of a given type:

```
// Create three App\User instances...
$users = factory(App\User::class, 3)->make();

// Create an App\User "admin" instance...
$user = factory(App\User::class, 'admin')->make();

// Create three App\User "admin" instances...
$users = factory(App\User::class, 'admin', 3)->make();
```

Persisting Factory Models

The `create` method not only creates the model instances, but also saves them to the database using Eloquent's `save` method:

```
public function testDatabase()
{
    $user = factory(App\User::class)->create();

    // Use model in tests...
}
```

Again, you may override attributes on the model by passing an array to the `create` method:

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
   ]);
```

Adding Relations To Models

You may even persist multiple models to the database. In this example, we'll even attach a relation to the created models. When using the `create` method to create multiple models, an Eloquent collection instance is returned, allowing you to use any of the convenient functions provided by the collection, such as `each`:

```
$users = factory(App\User::class, 3)
           ->create()
           ->each(function($u) {
                $u->posts()->save(factory(App\Post::class)->make());
            });
```

Mocking

Mocking Events

If you are making heavy use of Laravel's event system, you may wish to silence or mock certain events while testing. For example, if you are testing user registration, you probably do not want all of a `UserRegistered` event's handlers firing, since these may send "welcome" e-mails, etc.

Laravel provides a convenient `expectsEvents` method that verifies the expected events are fired, but prevents any handlers for those events from running:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->expectsEvents(App\Events\UserRegistered::class);
```

```
        // Test user registration code...
    }
}
```

If you would like to prevent all event handlers from running, you may use the `withoutEvents` method:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->withoutEvents();

        // Test user registration code...
    }
}
```

Mocking Jobs

Sometimes, you may wish to simply test that specific jobs are dispatched by your controllers when making requests to your application. This allows you to test your routes / controllers in isolation - set apart from your job's logic. Of course, you can then test the job itself in a separate test class.

Laravel provides a convenient `expectsJobs` method that will verify that the expected jobs are dispatched, but the job itself will not be executed:

```
<?php

class ExampleTest extends TestCase
{
    public function testPurchasePodcast()
    {
        $this->expectsJobs(App\Jobs\PurchasePodcast::class);

        // Test purchase podcast code...
    }
}
```

> **Note:** This method only detects jobs that are dispatched via the `DispatchesJobs` trait's dispatch methods. It does not detect jobs that are sent directly to `Queue::push`.

Mocking Facades

When testing, you may often want to mock a call to a Laravel facade. For example, consider the following controller action:

```
<?php

namespace App\Http\Controllers;

use Cache;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

We can mock the call to the `Cache` facade by using the `shouldReceive` method, which will return an instance of a Mockery mock. Since facades are actually resolved and managed by the Laravel service container, they have

much more testability than a typical static class. For example, let's mock our call to the `Cache` facade:

```
<?php

class FooTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

        $this->visit('/users')->see('value');
    }
}
```

> **Note:** You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP helper methods such as `call` and `post` when running your test.

Services

Validation

Introduction

Laravel provides several different approaches to validate your application's incoming data. By default, Laravel's base controller class uses a `ValidatesRequests` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user.

Defining The Routes

First, let's assume we have the following routes defined in our `app/Http/routes.php` file:

```
// Display a form to create a blog post...
Route::get('post/create', 'PostController@create');

// Store a new blog post...
Route::post('post', 'PostController@store');
```

Of course, the `GET` route will display a form for the user to create a new blog post, while the `POST` route will store the new blog post in the database.

Creating The Controller

Next, let's take a look at a simple controller that handles these routes. We'll leave the `store` method empty for now:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
```

```
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

Writing The Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. If you examine your application's base controller (`App\Http\Controllers\Controller`) class, you will see that the class uses a `ValidatesRequests` trait. This trait provides a convenient `validate` method in all of your controllers.

The `validate` method accepts an incoming HTTP request and a set of validation rules. If the validation rules pass, your code will keep executing normally; however, if validation fails, an exception will be thrown and the proper error response will automatically be sent back to the user. In the case of a traditional HTTP request, a redirect response will be generated, while a JSON response will be sent for AJAX requests.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

```
/**
 * Store a new blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid, store in database...
}
```

As you can see, we simply pass the incoming HTTP request and desired validation rules into the `validate` method. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

A Note On Nested Attributes

If your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request parameters do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our `GET` route. This is because Laravel will always check for errors in the session data, and automatically bind them to the view if

they are available. **So, it is important to note that an `$errors` variable will always be available in all of your views on every request**, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

So, in our example, the user will be redirected to our controller's `create` method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

Customizing The Flashed Error Format

If you wish to customize the format of the validation errors that are flashed to the session when validation fails, override the `formatValidationErrors` on your base controller. Don't forget to import the `Illuminate\Contracts\Validation\Validator` class at the top of the file:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Contracts\Validation\Validator;
use Illuminate\Routing\Controller as BaseController;
use Illuminate\Foundation\Validation\ValidatesRequests;

abstract class Controller extends BaseController
{
    use DispatchesJobs, ValidatesRequests;

    /**
     * {@inheritdoc}
     */
    protected function formatValidationErrors(Validator $validator)
    {
        return $validator->errors()->all();
    }
}
```

AJAX Requests & Validation

In this example, we used a traditional form to send data to the application. However, many applications use AJAX requests. When using the `validate` method during an AJAX request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

Other Validation Approaches

Manually Creating Validators

If you do not want to use the `ValidatesRequests` trait's `validate` method, you may create a validator instance manually using the `Validator` facade. The `make` method on the facade generates a new validator instance:

```
<?php

namespace App\Http\Controllers;
```

```
use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Store the blog post...
    }
}
```

The first argument passed to the `make` method is the data under validation. The second argument is the validation rules that should be applied to the data.

After checking if the request failed to pass validation, you may use the `withErrors` method to flash the error messages to the session. When using this method, the `$errors` variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The `withErrors` method accepts a validator, a `MessageBag`, or a PHP `array`.

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors, allowing you to retrieve the error messages for a specific form. Simply pass a name as the second argument to `withErrors`:

```
return redirect('register')
            ->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
{{ $errors->login->first('email') }}
```

After Validation Hook

The validator also allows you to attach callbacks to be run after validation is completed. This allows you to easily perform further validation and even add more error messages to the message collection. To get started, use the `after` method on a validator instance:

```
$validator = Validator::make(...);

$validator->after(function($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

Form Request Validation

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom

request classes that contain validation logic. To create a form request class, use the `make:request` Artisan CLI command:

```
php artisan make:request StoreBlogPostRequest
```

The generated class will be placed in the `app/Http/Requests` directory. Let's add a few validation rules to the `rules` method:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method is called, meaning you do not need to clutter your controller with any validation logic:

```
/**
 * Store the incoming blog post.
 *
 * @param  StoreBlogPostRequest  $request
 * @return Response
 */
public function store(StoreBlogPostRequest $request)
{
    // The incoming request is valid...
}
```

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an AJAX request, a HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may check if the authenticated user actually has the authority to update a given resource. For example, if a user is attempting to update a blog post comment, do they actually own that comment? For example:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $commentId = $this->route('comment');

    return Comment::where('id', $commentId)
                ->where('user_id', Auth::id())->exists();
}
```

Note the call to the `route` method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the `{comment}` parameter in the example below:

```
Route::post('comment/{comment}');
```

If the `authorize` method returns `false`, a HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to have authorization logic in another part of your application, simply return `true` from the `authorize` method:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

Customizing The Flashed Error Format

If you wish to customize the format of the validation errors that are flashed to the session when validation fails, override the `formatErrors` on your base request (`App\Http\Requests\Request`). Don't forget to import the `Illuminate\Contracts\Validation\Validator` class at the top of the file:

```
/**
 * {@inheritdoc}
 */
protected function formatErrors(Validator $validator)
{
    return $validator->errors()->all();
}
```

Customizing The Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}
```

Working With Error Messages

After calling the `errors` method on a `Validator` instance, you will receive an `Illuminate\Support\MessageBag` instance, which has a variety of convenient methods for working with error messages.

Retrieving The First Error Message For A Field

To retrieve the first error message for a given field, use the `first` method:

```
$messages = $validator->errors();

echo $messages->first('email');
```

Retrieving All Error Messages For A Field

If you wish to simply retrieve an array of all of the messages for a given field, use the `get` method:

```
foreach ($messages->get('email') as $message) {
    //
}
```

Retrieving All Error Messages For All Fields

To retrieve an array of all messages for all fields, use the `all` method:

```
foreach ($messages->all() as $message) {
```

```
    //
}
```

Determining If Messages Exist For A Field

```
if ($messages->has('email')) {
    //
}
```

Retrieving An Error Message With A Format

```
echo $messages->first('email', '<p>:message</p>');
```

Retrieving All Error Messages With A Format

```
foreach ($messages->all('<li>:message</li>') as $message) {
    //
}
```

Custom Error Messages

If needed, you may use custom error messages for validation instead of the defaults. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

In this example, the `:attribute` place-holder will be replaced by the actual name of the field under validation. You may also utilize other place-holders in validation messages. For example:

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following types: :values',
];
```

Specifying A Custom Message For A Given Attribute

Sometimes you may wish to specify a custom error messages only for a specific field. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

Specifying Custom Messages In Language Files

In many cases, you may wish to specify your attribute specific custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

Available Validation Rules

Below is a list of all available validation rules and their function:

accepted

The field under validation must be *yes*, *on*, *1*, or *true*. This is useful for validating "Terms of Service" acceptance.

active_url

The field under validation must be a valid URL according to the `checkdnsrr` PHP function.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

alpha

The field under validation must be entirely alphabetic characters.

alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

alpha_num

The field under validation must be entirely alpha-numeric characters.

array

The field under validation must be a PHP `array`.

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

date

The field under validation must be a valid date according to the `strtotime` PHP function.

date_format:*format*

The field under validation must match the given *format*. The format will be evaluated using the PHP `date_parse_from_format` function. You should use **either** `date` or `date_format` when validating a field, not both.

different:*field*

The field under validation must have a different value than *field*.

digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

digits_between:*min,max*

The field under validation must have a length between the given *min* and *max*.

email

The field under validation must be formatted as an e-mail address.

exists:*table,column*

The field under validation must exist on a given database table.

Basic Usage Of Exists Rule

```
'state' => 'exists:states'
```

Specifying A Custom Column Name

```
'state' => 'exists:states,abbreviation'
```

You may also specify more conditions that will be added as "where" clauses to the query:

```
'email' => 'exists:staff,email,account_id,1'
```

You may also pass `NULL` or `NOT_NULL` to the "where" clause:

```
'email' => 'exists:staff,email,deleted_at,NULL'

'email' => 'exists:staff,email,deleted_at,NOT_NULL'
```

image

The file under validation must be an image (jpeg, png, bmp, gif, or svg)

in:*foo,bar,...*

The field under validation must be included in the given list of values.

integer

The field under validation must be an integer.

ip

The field under validation must be an IP address.

json

The field under validation must be a valid JSON string.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

mimes:*foo,bar,...*

The file under validation must have a MIME type corresponding to one of the listed extensions.

Basic Usage Of MIME Rule

```
'photo' => 'mimes:jpeg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates against the MIME type of the file by reading the file's contents and guessing its MIME type.

A full listing of MIME types and their corresponding extensions may be found at the following location: http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

not_in:*foo,bar,...*

The field under validation must not be included in the given list of values.

numeric

The field under validation must be numeric.

regex:*pattern*

The field under validation must match the given regular expression.

Note: When using the `regex` pattern, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

required

The field under validation must be present in the input data and not empty. A field is considered "empty" is one of the following conditions are true:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.

required_if:*anotherfield,value,...*

The field under validation must be present if the *anotherfield* field is equal to any *value*.

required_unless:*anotherfield,value,...*

The field under validation must be present unless the *anotherfield* field is equal to any *value*.

required_with:*foo,bar,...*

The field under validation must be present *only if* any of the other specified fields are present.

required_with_all:*foo,bar,...*

The field under validation must be present *only if* all of the other specified fields are present.

required_without:*foo,bar,...*

The field under validation must be present *only when* any of the other specified fields are not present.

required_without_all:*foo,bar,...*

The field under validation must be present *only when* all of the other specified fields are not present.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value. For files, *size* corresponds to the file size in kilobytes.

string

The field under validation must be a string.

timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

unique:*table,column,except,idColumn*

The field under validation must be unique on a given database table. If the `column` option is not specified, the field name will be used.

Specifying A Custom Column Name:

```
'email' => 'unique:users,email_address'
```

Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. As seen above, setting `unique:users` as a validation rule will use the default database connection to query the database. To override this, specify the connection followed by the table name using "dot" syntax:

```
'email' => 'unique:connection.users,email_address'
```

Forcing A Unique Rule To Ignore A Given ID:

Sometimes, you may wish to ignore a given ID during the unique check. For example, consider an "update profile" screen that includes the user's name, e-mail address, and location. Of course, you will want to verify that the e-mail address is unique. However, if the user only changes the name field and not the e-mail field, you do not want a validation error to be thrown because the user is already the owner of the e-mail address. You only want to throw a validation error if the user provides an e-mail address that is already used by a different user. To tell the unique rule to ignore the user's ID, you may pass the ID as the third parameter:

```
'email' => 'unique:users,email_address,'.$user->id
```

If your table uses a primary key column name other than `id`, you may specify it as the fourth parameter:

```
'email' => 'unique:users,email_address,'.$user->id.',user_id'
```

Adding Additional Where Clauses:

You may also specify more conditions that will be added as "where" clauses to the query:

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

In the rule above, only rows with an `account_id` of `1` would be included in the unique check.

url

The field under validation must be a valid URL according to PHP's `filter_var` function.

Conditionally Adding Rules

In some situations, you may wish to run validation checks against a field **only** if that field is present in the input array. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game re-sell shop, or maybe they just enjoy collecting. To conditionally add this requirement, we can

use the `sometimes` method on the `Validator` instance.

```
$v->sometimes('reason', 'required|max:500', function($input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is the rules we want to add. If the `Closure` passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$v->sometimes(['reason', 'cost'], 'required', function($input) {
    return $input->games >= 100;
});
```

> **Note:** The `$input` parameter passed to your `Closure` will be an instance of `Illuminate\Support\Fluent` and may be used to access your input and files.

Custom Validation Rules

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using the `extend` method on the `Validator` facade. Let's use this method within a service provider to register a custom validation rule:

```
<?php

namespace App\Providers;

use Validator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

The custom validator Closure receives four arguments: the name of the `$attribute` being validated, the `$value` of the attribute, an array of `$parameters` passed to the rule, and the `Validator` instance.

You may also pass a class and method to the `extend` method instead of a Closure:

```
Validator::extend('foo', 'FooValidator@validate');
```

Defining The Error Message

You will also need to define an error message for your custom rule. You can do so either using an inline custom message array or by adding an entry in the validation language file. This message should be placed in the first level of the array, not within the `custom` array, which is only for attribute-specific error messages:

```
"foo" => "Your input was invalid!",
```

```
"accepted" => "The :attribute must be accepted.",

// The rest of the validation error messages...
```

When creating a custom validation rule, you may sometimes need to define custom place-holder replacements for error messages. You may do so by creating a custom Validator as described above then making a call to the `replacer` method on the `Validator` facade. You may do this within the `boot` method of a service provider:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Validator::extend(...);

    Validator::replacer('foo', function($message, $attribute, $rule, $parameters) {
        return str_replace(...);
    });
}
```

Implicit Extensions

By default, when an attribute being validated is not present or contains an empty value as defined by the `required` rule, normal validation rules, including custom extensions, are not run. For example, the `integer` rule will not be run against a `null` value:

```
$rules = ['count' => 'integer'];

$input = ['count' => null];

Validator::make($input, $rules)->passes(); // true
```

For a rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create such an "implicit" extension, use the `Validator::extendImplicit()` method:

```
Validator::extendImplicit('foo', function($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

> **Note:** An "implicit" extension only *implies* that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Database

Database: Getting Started

Introduction

Laravel makes connecting with databases and running queries extremely simple across a variety of database back-ends using either raw SQL, the fluent query builder, and the Eloquent ORM. Currently, Laravel supports four database systems:

- MySQL
- Postgres
- SQLite
- SQL Server

Configuration

Laravel makes connecting with databases and running queries extremely simple. The database configuration for your application is located at `config/database.php`. In this file you may define all of your database connections, as well as specify which connection should be used by default. Examples for all of the supported database systems are provided in this file.

By default, Laravel's sample environment configuration is ready to use with Laravel Homestead, which is a convenient virtual machine for doing Laravel development on your local machine. Of course, you are free to modify this configuration as needed for your local database.

Read / Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
],
```

Note that two keys have been added to the configuration array: `read` and `write`. Both of these keys have array values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` array.

So, we only need to place items in the `read` and `write` arrays if we wish to override the values in the main array. So, in this case, `192.168.1.1` will be used as the "read" connection, while `192.168.1.2` will be used as the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` array will be

shared across both connections.

Running Raw SQL Queries

Once you have configured your database connection, you may run queries using the `DB` facade. The `DB` facade provides methods for each type of query: `select`, `update`, `insert`, `delete`, and `statement`.

Running A Select Query

To run a basic query, we can use the `select` method on the `DB` facade:

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);

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

The first argument passed to the `select` method is the raw SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the `where` clause constraints. Parameter binding provides protection against SQL injection.

The `select` method will always return an `array` of results. Each result within the array will be a PHP `stdClass` object, allowing you to access the values of the results:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Using Named Bindings

Instead of using `?` to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Running An Insert Statement

To execute an `insert` statement, you may use the `insert` method on the `DB` facade. Like `select`, this method takes the raw SQL query as its first argument, and bindings as the second argument:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Running An Update Statement

The `update` method should be used to update existing records in the database. The number of rows affected by the statement will be returned by the method:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

Running A Delete Statement

The `delete` method should be used to delete records from the database. Like `update`, the number of rows deleted will be returned:

```
$deleted = DB::delete('delete from users');
```

Running A General Statement

Some database statements should not return any value. For these types of operations, you may use the `statement` method on the `DB` facade:

```
DB::statement('drop table users');
```

Listening For Query Events

If you would like to receive each SQL query executed by your application, you may use the `listen` method. This method is useful for logging queries or debugging. You may register your query listener in a service provider:

```
<?php

namespace App\Providers;

use DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function($sql, $bindings, $time) {
            //
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Database Transactions

To run a set of operations within a database transaction, you may use the `transaction` method on the `DB` facade. If an exception is thrown within the transaction `Closure`, the transaction will automatically be rolled back. If the `Closure` executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the `transaction` method:

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the `beginTransaction` method on the `DB` facade:

```
DB::beginTransaction();
```

You can rollback the transaction via the `rollBack` method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the `commit` method:

```
DB::commit();
```

> **Note:** Using the `DB` facade's transaction methods also controls transactions for the query builder and Eloquent ORM.

Using Multiple Database Connections

When using multiple connections, you may access each connection via the `connection` method on the `DB` facade. The `name` passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file:

```
$users = DB::connection('foo')->select(...);
```

You may also access the raw, underlying PDO instance using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Database

Database: Query Builder

Introduction

The database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application, and works on all supported database systems.

> **Note:** The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean strings being passed as bindings.

Retrieving Results

Retrieving All Rows From A Table

To begin a fluent query, use the `table` method on the `DB` facade. The `table` method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally get the results. In this example, let's just `get` all records from a table:

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

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

Like raw queries, the `get` method returns an `array` of results where each result is an instance of the PHP `StdClass` object. You may access each column's value by accessing the column as a property of the object:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from the database table, you may use the `first` method. This method will return a single `StdClass` object:

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

If you don't even need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Chunking Results From A Table

If you need to work with thousands of database records, consider using the `chunk` method. This method retrieves a small "chunk" of the results at a time, and feeds each chunk into a `Closure` for processing. This method is very useful for writing Artisan commands that process thousands of records. For example, let's work with the entire `users` table in chunks of 100 records at a time:

```
DB::table('users')->chunk(100, function($users) {
    foreach ($users as $user) {
        //
    }
});
```

You may stop further chunks from being processed by returning `false` from the `Closure`:

```
DB::table('users')->chunk(100, function($users) {
    // Process the records...

    return false;
});
```

Retrieving A List Of Column Values

If you would like to retrieve an array containing the values of a single column, you may use the `lists` method. In this example, we'll retrieve an array of role titles:

```
$titles = DB::table('roles')->lists('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may also specify a custom key column for the returned array:

```
$roles = DB::table('roles')->lists('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

Aggregates

The query builder also provides a variety of aggregate methods, such as `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses to build your query:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Selects

Specifying A Select Clause

Of course, you may not always want to select all columns from a database table. Using the `select` method, you can specify a custom `select` clause for the query:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

The `distinct` method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the `addSelect` method:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Raw Expressions

Sometimes you may need to use a raw expression in a query. These expressions will be injected into the query as strings, so be careful not to create any SQL injection points! To create a raw expression, you may use the `DB::raw` method:

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

Joins

Inner Join Statement

The query builder may also be used to write join statements. To perform a basic SQL "inner join", you may use the `join` method on a query builder instance. The first argument passed to the `join` method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. Of course, as you can see, you can join to multiple tables in a single query:

```
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join Statement

If you would like to perform a "left join" instead of an "inner join", use the `leftJoin` method. The `leftJoin` method has the same signature as the `join` method:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Advanced Join Statements

You may also specify more advanced join clauses. To get started, pass a `Closure` as the second argument into the `join` method. The `Closure` will receive a `JoinClause` object which allows you to specify constraints on the `join` clause:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

If you would like to use a "where" style clause on your joins, you may use the `where` and `orWhere` methods on a join. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Unions

The query builder also provides a quick way to "union" two queries together. For example, you may create an initial query, and then use the `union` method to union it with a second query:

```
$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

The `unionAll` method is also available and has the same method signature as `union`.

Where Clauses

Simple Where Clauses

To add `where` clauses to the query, use the `where` method on a query builder instance. The most basic call to `where` requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to evaluate against the column.

For example, here is a query that verifies the value of the "votes" column is equal to 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

For convenience, if you simply want to verify that a column is equal to a given value, you may pass the value directly as the second argument to the `where` method:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Of course, you may use a variety of other operators when writing a `where` clause:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

Or Statements

You may chain where constraints together, as well as add `or` clauses to the query. The `orWhere` method accepts the same arguments as the `where` method:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Additional Where Clauses

whereBetween

The `whereBetween` method verifies that a column's value is between two values:

```
$users = DB::table('users')
                    ->whereBetween('votes', [1, 100])->get();
```

whereNotBetween

The `whereNotBetween` method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

whereIn / whereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

The `whereNotIn` method verifies that the given column's value is **not** contained in the given array:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

whereNull / whereNotNull

The `whereNull` method verifies that the value of the given column is `NULL`:

```
$users = DB::table('users')
                    ->whereNull('updated_at')
                    ->get();
```

The `whereNotNull` method verifies that the column's value is **not** `NULL`:

```
$users = DB::table('users')
                    ->whereNotNull('updated_at')
                    ->get();
```

Advanced Where Clauses

Parameter Grouping

Sometimes you may need to create more advanced where clauses such as "where exists" or nested parameter groupings. The Laravel query builder can handle these as well. To get started, let's look at an example of grouping constraints within parenthesis:

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function ($query) {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

As you can see, passing `Closure` into the `orWhere` method instructs the query builder to begin a constraint group. The `Closure` will receive a query builder instance which you can use to set the constraints that should be contained within the parenthesis group. The example above will produce the following SQL:

```
select * from users where name = 'John' or (votes > 100 and title <> 'Admin')
```

Exists Statements

The `whereExists` method allows you to write `where exist` SQL clauses. The `whereExists` method accepts a

`Closure` argument, which will receive a query builder instance allowing you to define the query that should be placed inside of the "exists" clause:

```
DB::table('users')
            ->whereExists(function ($query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

The query above will produce the following SQL:

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

Ordering, Grouping, Limit, & Offset

orderBy

The `orderBy` method allows you to sort the result of the query by a given column. The first argument to the `orderBy` method should be the column you wish to sort by, while the second argument controls the direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

groupBy / having / havingRaw

The `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

The `havingRaw` method may be used to set a raw string as the value of the `having` clause. For example, we can find all of the departments with sales greater than $2,500:

```
$users = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > 2500')
                ->get();
```

skip / take

To limit the number of results returned from the query, or to skip a given number of results in the query (`OFFSET`), you may use the `skip` and `take` methods:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Inserts

The query builder also provides an `insert` method for inserting records into the database table. The `insert` method accepts an array of column names and values to insert:

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

You may even insert several records into the table with a single call to `insert` by passing an array of arrays. Each array represents a row to be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

> **Note:** When using PostgreSQL the insertGetId method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the sequence name as the second parameter to the `insertGetId` method.

Updates

Of course, in addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs containing the columns to be updated. You may constrain the `update` query using `where` clauses:

```
DB::table('users')
            ->where('id', 1)
            ->update(['votes' => 1]);
```

Increment / Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. This is simply a short-cut, providing a more expressive and terse interface compared to manually writing the `update` statement.

Both of these methods accept at least one argument: the column to modify. A second argument may optionally be passed to control the amount by which the column should be incremented / decremented.

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

You may also specify additional columns to update during the operation:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Deletes

Of course, the query builder may also be used to delete records from the table via the `delete` method:

```
DB::table('users')->delete();
```

You may constrain `delete` statements by adding `where` clauses before calling the `delete` method:

```
DB::table('users')->where('votes', '<', 100)->delete();
```

If you wish to truncate the entire table, which will remove all rows and reset the auto-incrementing ID to zero, you may use the `truncate` method:

```
DB::table('users')->truncate();
```

Pessimistic Locking

The query builder also includes a few functions to help you do "pessimistic locking" on your `select` statements. To run the statement with a "shared lock", you may use the `sharedLock` method on a query. A shared lock prevents the selected rows from being modified until your transaction commits:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the rows from being modified or from being selected with another shared lock:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

Database

Database: Migrations

Introduction

Migrations are like version control for your database, allowing a team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables. It shares the same expressive, fluent API across all of Laravel's supported database systems.

Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Laravel to determine the order of the migrations.

The `--table` and `--create` options may also be used to indicate the name of the table and whether the migration will be creating a new table. These options simply pre-fill the generated migration stub file with the specified table:

```
php artisan make:migration add_votes_to_users_table --table=users
```

```
php artisan make:migration create_users_table --create=users
```

If you would like to specify a custom output path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The provided path should be relative to your application's base path.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should simply reverse the operations performed by the `up` method.

Within both of these methods you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, check out its documentation. For example, let's look at a sample migration that creates a `flights` table:

```
<?php
```

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Running Migrations

To run all outstanding migrations for your application, use the `migrate` Artisan command. If you are using the Homestead virtual machine, you should run this command from within your VM:

```
php artisan migrate
```

If you receive a "class not found" error when running migrations, try running the `composer dump-autoload` command and re-issuing the migrate command.

Forcing Migrations To Run In Production

Some migration operations are destructive, meaning 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 these commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To rollback the latest migration "operation", you may use the `rollback` command. Note that this rolls back the last "batch" of migrations that ran, which may include multiple migration files:

```
php artisan migrate:rollback
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Rollback / Migrate In Single Command

The `migrate:refresh` command will first roll back all of your database migrations, and then run the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

php artisan migrate:refresh --seed
```

Writing Migrations

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 used to define the new table:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Of course, when creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
    //
}
```

Connection & Storage Engine

If you want to perform a schema operation on a database connection that is not your default connection, use the `connection` method:

```
Schema::connection('foo')->create('users', function ($table) {
    $table->increments('id');
});
```

To set the storage engine for a table, set the `engine` property on the schema builder:

```
Schema::create('users', function ($table) {
    $table->engine = 'InnoDB';

    $table->increments('id');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Creating Columns

To update an existing table, we will use the `table` method on the `Schema` facade. Like the `create` method, the `table` method accepts two arguments: the name of the table and a `Closure` that receives a `Blueprint` instance we can use to add columns to the table:

```
Schema::table('users', function ($table) {
    $table->string('email');
});
```

Available Column Types

Of course, the schema builder contains a variety of column types that you may use when building your tables:

Command	Description
`$table->bigIncrements('id');`	Incrementing ID (primary key) using a "UNSIGNED BIG INTEGER" equivalent.
`$table->bigInteger('votes');`	BIGINT equivalent for the database.
`$table->binary('data');`	BLOB equivalent for the database.
`$table->boolean('confirmed');`	BOOLEAN equivalent for the database.
`$table->char('name', 4);`	CHAR equivalent with a length.
`$table->date('created_at');`	DATE equivalent for the database.
`$table->dateTime('created_at');`	DATETIME equivalent for the database.
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale.
`$table->double('column', 15, 8);`	DOUBLE equivalent with precision, 15 digits in total and 8 after the decimal point.
`$table->enum('choices', ['foo', 'bar']);`	ENUM equivalent for the database.
`$table->float('amount');`	FLOAT equivalent for the database.
`$table->increments('id');`	Incrementing ID (primary key) using a "UNSIGNED INTEGER" equivalent.
`$table->integer('votes');`	INTEGER equivalent for the database.
`$table->json('options');`	JSON equivalent for the database.
`$table->jsonb('options');`	JSONB equivalent for the database.
`$table->longText('description');`	LONGTEXT equivalent for the database.
`$table->mediumInteger('numbers');`	MEDIUMINT equivalent for the database.
`$table->mediumText('description');`	MEDIUMTEXT equivalent for the database.
`$table->morphs('taggable');`	Adds INTEGER `taggable_id` and STRING `taggable_type`.
`$table->nullableTimestamps();`	Same as `timestamps()`, except allows NULLs.
`$table->rememberToken();`	Adds `remember_token` as VARCHAR(100) NULL.
`$table->smallInteger('votes');`	SMALLINT equivalent for the database.
`$table->softDeletes();`	Adds `deleted_at` column for soft deletes.
`$table->string('email');`	VARCHAR equivalent column.
`$table->string('name', 100);`	VARCHAR equivalent with a length.
`$table->text('description');`	TEXT equivalent for the database.
`$table->time('sunrise');`	TIME equivalent for the database.
`$table->tinyInteger('numbers');`	TINYINT equivalent for the database.
`$table->timestamp('added_on');`	TIMESTAMP equivalent for the database.
`$table->timestamps();`	Adds `created_at` and `updated_at` columns.
`$table->uuid('id');`	UUID equivalent for the database.

Column Modifiers

In addition to the column types listed above, there are several other column "modifiers" which you may use while adding the column. For example, to make the column "nullable", you may use the `nullable` method:

```
Schema::table('users', function ($table) {
    $table->string('email')->nullable();
});
```

Below is a list of all the available column modifiers. This list does not include the index modifiers:

Modifier	Description
`->first()`	Place the column "first" in the table (MySQL Only)
`->after('column')`	Place the column "after" another column (MySQL Only)
`->nullable()`	Allow NULL values to be inserted into the column
`->default($value)`	Specify a "default" value for the column
`->unsigned()`	Set `integer` columns to `UNSIGNED`

Modifying Columns

Prerequisites

Before modifying a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file. The Doctrine DBAL library is used to determine the current state of the column and create the SQL queries needed to make the specified adjustments to the column.

Updating Column Attributes

The `change` method allows you to modify an existing column to a new type, or modify the column's attributes. For example, you may wish to increase the size of a string column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50:

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->change();
});
```

We could also modify a column to be nullable:

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->nullable()->change();
});
```

Renaming Columns

To rename a column, you may use the `renameColumn` method on the Schema builder. Before renaming a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file:

```
Schema::table('users', function ($table) {
    $table->renameColumn('from', 'to');
});
```

> **Note:** Renaming columns in a table with a `enum` column is not currently supported.

Dropping Columns

To drop a column, use the `dropColumn` method on the Schema builder:

```
Schema::table('users', function ($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 ($table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

> **Note:** Before dropping columns from a SQLite database, you will need to add the `doctrine/dbal` dependency to your `composer.json` file and run the `composer update` command in your terminal to install the library.

> **Note:** Dropping or modifying multiple columns within a single migration while using a SQLite database is not supported.

Creating Indexes

The schema builder supports several types of indexes. First, let's look at an example that specifies a column's values should be unique. To create the index, we can simply chain the `unique` method onto the column definition:

```
$table->string('email')->unique();
```

Alternatively, you may create the index after defining the column. For example:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound index:

```
$table->index(['account_id', 'created_at']);
```

Available Index Types

Command	Description
`$table->primary('id');`	Add a primary key.
`$table->primary(['first', 'last']);`	Add composite keys.
`$table->unique('email');`	Add a unique index.
`$table->index('state');`	Add a basic index.

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns a reasonable name to the indexes. Simply concatenate the table name, the name of the indexed column, and the index type. Here are some examples:

Command	Description
`$table->dropPrimary('users_id_primary');`	Drop a primary key from the "users" table.
`$table->dropUnique('users_email_unique');`	Drop a unique index from the "users" table.
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.

If you pass an array of columns into a method that drops indexes, the conventional index name will be generated based on the table name, columns and key type.

```
Schema::table('geo', function ($table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

```
Schema::table('posts', function ($table) {
    $table->integer('user_id')->unsigned();

    $table->foreign('user_id')->references('id')->on('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

To drop a foreign key, you may use the `dropForeign` method. Foreign key constraints use the same naming convention as indexes. So, we will concatenate the table name and the columns in the constraint then suffix the name with "_foreign":

```
$table->dropForeign('posts_user_id_foreign');
```

Or you may pass an array value which will automatically use the conventional constraint name when dropping:

```
$table->dropForeign(['user_id']);
```

Database

Database: Seeding

Introduction

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in `database/seeds`. Seed classes may have any name you wish, but probably should follow some sensible convention, such as `UsersTableSeeder`, etc. By default, a `DatabaseSeeder` class is defined for you. From this class, you may use the `call` method to run other seed classes, allowing you to control the seeding order.

Writing Seeders

To generate a seeder, you may issue the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeders` directory:

```
php artisan make:seeder UsersTableSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the query builder to manually insert data or you may use Eloquent model factories.

As an example, let's modify the `DatabaseSeeder` class which is included with a default installation of Laravel. Let's add a database insert statement to the `run` method:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use model factories to conveniently generate large amounts of database records. First, review the model factory documentation to learn how to define your factories. Once you have defined your factories, you may use the `factory` helper function to insert records into your database.

For example, let's create 50 users and attach a relationship to each user:

```
/**
 * Run the database seeds.
 *
```

```
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you may use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes overwhelmingly large. Simply pass the name of the seeder class you wish to run:

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    Model::unguard();

    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);

    Model::reguard();
}
```

Running Seeders

Once you have written your seeder classes, you may use the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `DatabaseSeeder` class, which may be used to call other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed
```

```
php artisan db:seed --class=UserTableSeeder
```

You may also seed your database using the `migrate:refresh` command, which will also rollback and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:refresh --seed
```

Eloquent ORM

Eloquent: Getting Started

Introduction

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

Before getting started, be sure to configure a database connection in `config/database.php`. For more information on configuring your database, check out the documentation.

Defining Models

To get started, let's create an Eloquent model. Models typically live in the `app` directory, but you are free to place them anywhere that can be auto-loaded according to your `composer.json` file. All Eloquent models extend `Illuminate\Database\Eloquent\Model` class.

The easiest way to create a model instance is using the `make:model` Artisan command:

```
php artisan make:model User
```

If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model User --migration
```

```
php artisan make:model User -m
```

Eloquent Model Conventions

Now, let's look at an example `Flight` model class, which we will use to retrieve and store information from our `flights` database table:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

Note that we did not tell Eloquent which table to use for our `Flight` model. The "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `Flight` model stores records in the `flights` table. You may specify a custom table by defining a `table` property on your model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each table has a primary key column named `id`. You may define a `$primaryKey` property to override this convention.

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to `false`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

If you need to customize the format of your timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database, as well as their format when the model is serialized to an array or JSON:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Database Connection

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the `$connection` property:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Retrieving Multiple Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Show a list of all available flights.
     *
     * @return Response
     */
    public function index()
    {
        $flights = Flight::all();

        return view('flight.index', ['flights' => $flights]);
    }
}
```

Accessing Column Values

If you have an Eloquent model instance, you may access the column values of the model by accessing the corresponding property. For example, let's loop through each `Flight` instance returned by our query and echo the value of the `name` column:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Adding Additional Constraints

The Eloquent `all` method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the `get` method to retrieve the results:

```
$flights = App\Flight::where('active', 1)
               ->orderBy('name', 'desc')
               ->take(10)
               ->get();
```

> **Note:** Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

Collections

For Eloquent methods like `all` and `get` which retrieve multiple results, an instance of `Illuminate\Database\Eloquent\Collection` will be returned. The `Collection` class provides a variety of helpful methods for working with your Eloquent results. Of course, you may simply loop over this collection like an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

If you need to process thousands of Eloquent records, use the `chunk` command. The `chunk` method will retrieve a "chunk" of Eloquent models, feeding them to a given `Closure` for processing. Using the `chunk` method will conserve memory when working with large result sets:

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called for each chunk that is retrieved from the database.

Retrieving Single Models / Aggregates

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using `find` and `first`. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query. However, if no result is found, a `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a `404` HTTP response is automatically sent back to the user, so it is not necessary to write explicit checks to return `404` responses when using these methods:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

Retrieving Aggregates

Of course, you may also use `count`, `sum`, `max`, and other aggregate functions provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Basic Inserts

To create a new record in the database, simply create a new model instance, set attributes on the model, then call the `save` method:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

In this example, we simply assign the `name` parameter from the incoming HTTP request to the `name` attribute of the `App\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Basic Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it, set any attributes you wish to update, and then call the `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Updates can also be performed against any number of models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
App\Flight::where('active', 1)
          ->where('destination', 'San Diego')
          ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated.

Mass Assignment

You may also use the `create` method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a `fillable` or `guarded` attribute on the model, as all Eloquent models protect against mass-assignment.

A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then mapped onto your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database. The `create` method returns the saved model instance:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

While `$fillable` serves as a "white list" of attributes that should be mass assignable, you may also choose to use `$guarded`. The `$guarded` property should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, `$guarded` functions like a "black list". Of course, you should use either `$fillable` or `$guarded` - not both:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

In the example above, all attributes **except for `price`** will be mass assignable.

Other Creation Methods

There are two other methods you may use to create models by mass assigning attributes: `firstOrCreate` and `firstOrNew`. The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the given attributes.

The `firstOrNew` method, like `firstOrCreate` will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to call `save` manually to persist it:

```
// Retrieve the flight by the attributes, or create it if it doesn't exist...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// Retrieve the flight by the attributes, or instantiate a new instance...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);
```

Deleting Models

To delete a model, call the `delete` method on a model instance:

```
$flight = App\Flight::find(1);
```

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

Deleting An Existing Model By Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without retrieving it. To do so, call the `destroy` method:

```
App\Flight::destroy(1);

App\Flight::destroy([1, 2, 3]);

App\Flight::destroy(1, 2, 3);
```

Deleting Models By Query

Of course, you may also run a delete query on a set of models. In this example, we will delete all flights that are marked as inactive:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model and inserted into the database. If a model has a non-null `deleted_at` value, the model has been soft deleted. To enable soft deletes for a model, use the `Illuminate\Database\Eloquent\SoftDeletes` trait on the model and add the `deleted_at` column to your `$dates` property:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;

    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

Of course, you should add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. And, when querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, use the `trashed` method:

```
if ($flight->trashed()) {
    //
}
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to appear in a result set using the `withTrashed` method on the query:

```
$flights = App\Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

The `withTrashed` method may also be used on a relationship query:

```
$flight->history()->withTrashed()->get();
```

Where Clause Caveats

When adding `orWhere` clauses to your queries on soft deleted models, always use advance where clauses to logically group the `WHERE` clauses. For example:

```
User::where(function($query) {
        $query->where('name', '=', 'John')
              ->orWhere('votes', '>', 100);
        })
        ->get();
```

This will produce the following SQL:

```
select * from `users` where `users`.`deleted_at` is null and (`name` = 'John' or `votes` > 100)
```

If the `orWhere` clause is not grouped, it will produce the following SQL which will contain soft deleted records:

```
select * from `users` where `users`.`deleted_at` is null and `name` = 'John' or `votes` > 100
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = App\Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model into an active state, use the `restore` method on a model instance:

```
$flight->restore();
```

You may also use the `restore` method in a query to quickly restore multiple models:

```
App\Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

Like the `withTrashed` method, the `restore` method may also be used on relationships:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. To permanently remove a soft deleted model from the database, use the `forceDelete` method:

```
// Force deleting a single model instance...
$flight->forceDelete();

// Force deleting all related models...
$flight->history()->forceDelete();
```

Query Scopes

Scopes allow you to define common sets of constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, simply prefix an Eloquent model method with `scope`.

Scopes should always return a query builder instance:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}
```

Utilizing A Query Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you do not need to include the `scope` prefix when calling the method. You can even chain calls to various scopes, for example:

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope. Scope parameters should be defined after the `$query` argument:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

Now, you may pass the parameters when calling the scope:

```
$users = App\User::ofType('admin')->get();
```

Events

Eloquent models fire several events, allowing you to hook into various points in the model's lifecycle using the following methods: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`. Events allow you to easily execute code each time a specific model class is saved or updated in the database.

Basic Usage

Whenever a new model is saved for the first time, the `creating` and `created` events will fire. If a model already existed in the database and the `save` method is called, the `updating` / `updated` events will fire. However, in both cases, the `saving` / `saved` events will fire.

For example, let's define an Eloquent event listener in a service provider. Within our event listener, we will call the `isValid` method on the given model, and return `false` if the model is not valid. Returning `false` from an Eloquent event listener will cancel the `save` / `update` operation:

```
<?php

namespace App\Providers;

use App\User;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        User::creating(function ($user) {
            if ( ! $user->isValid()) {
                return false;
            }
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Eloquent ORM

Eloquent: Relationships

Introduction

Database tables are often related to one another. For example, a blog post may have many comments, or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports several different types of relationships:

- One To One
- One To Many
- Many To Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

Defining Relationships

Eloquent relationships are defined as functions on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful query builders, defining relationships as functions provides powerful method chaining and querying capabilities. For example:

```
$user->posts()->where('active', 1)->get();
```

But, before diving too deep into using relationships, let's learn how to define each type:

One To One

A one-to-one relationship is a very basic relation. For example, a `User` model might be associated with one `Phone`. To define this relationship, we place a `phone` method on the `User` model. The `phone` method should return the results of the `hasOne` method on the base Eloquent model class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
```

```
        return $this->hasOne('App\Phone');
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship functions as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent assumes the foreign key of the relationship based on the model name. In this case, the `Phone` model is automatically assumed to have a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the `id` column of the parent. In other words, Eloquent will look for the value of the user's `id` column in the `user_id` column of the `Phone` record. If you would like the relationship to use a value other than `id`, you may pass a third argument to the `hasOne` method specifying your custom key:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relation

So, we can access the `Phone` model from our `User`. Now, let's define a relationship on the `Phone` model that will let us access the `User` that owns the phone. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

In the example above, Eloquent will try to match the `user_id` from the `Phone` model to an `id` on the `User` model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. However, if the foreign key on the `Phone` model is not `user_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

One To Many

A "one-to-many" relationship is used to define relationships where a single model owns any amount of other models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by placing a function on your Eloquent model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column on the `Comment` model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with `_id`. So, for this example, Eloquent will assume the foreign key on the `Comment` model is `post_id`.

Once the relationship has been defined, we can access the collection of comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic properties", we can access relationship functions as if they were defined as properties on the model:

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Of course, since all relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the `comments` method and continuing to chain conditions onto the query:

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

Like the `hasOne` method, you may also override the foreign and local keys by passing additional arguments to the `hasMany` method:

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relation

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a `hasMany` relationship, define a relationship function on the child model which calls the `belongsTo` method:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Once the relationship has been defined, we can retrieve the `Post` model for a `Comment` by accessing the `post` "dynamic property":

```
$comment = App\Comment::find(1);

echo $comment->post->title;
```

In the example above, Eloquent will try to match the `post_id` from the `Comment` model to an `id` on the `Post` model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. However, if the foreign key on the `Comment` model is not `post_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

Many To Many

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin". To define this relationship, three database tables are needed: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names, and contains the `user_id` and `role_id` columns.

Many-to-many relationships are defined by writing a method that calls the `belongsToMany` method on the base Eloquent class. For example, let's define the `roles` method on our `User` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

Once the relationship is defined, you may access the user's roles using the `roles` dynamic property:

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

foreach ($user->roles as $role) {
    //
}
```

Of course, like all other relationship types, you may call the `roles` method to continue chaining query constraints onto the relationship:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

As mentioned previously, to determine the table name of the relationship's joining table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the `belongsToMany` method:

```
return $this->belongsToMany('App\Role', 'user_roles');
```

In addition to customizing the name of the joining table, you may also customize the column names of the keys on the table by passing additional arguments to the `belongsToMany` method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the inverse of a many-to-many relationship, you simply place another call to `belongsToMany` on your related model. To continue our user roles example, let's define the `users` method on the `Role` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

As you can see, the relationship is defined exactly the same as its `User` counterpart, with the exception of simply referencing the `App\User` model. Since we're reusing the `belongsToMany` method, all of the usual table and key customization options are available when defining the inverse of many-to-many relationships.

Retrieving Intermediate Table Columns

As you have already learned, working with many-to-many relations requires the presence of an intermediate table. Eloquent provides some very helpful ways of interacting with this table. For example, let's assume our `User` object has many `Role` objects that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

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

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notice that each `Role` model we retrieve is automatically assigned a `pivot` attribute. This attribute contains a model representing the intermediate table, and may be used like any other Eloquent model.

By default, only the model keys will be present on the `pivot` object. If your pivot table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

If you want your pivot table to have automatically maintained `created_at` and `updated_at` timestamps, use the `withTimestamps` method on the relationship definition:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Has Many Through

The "has-many-through" relationship provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a `Country` model might have many `Post` models through an intermediate `User` model. In this example, you could easily gather all blog posts for a given country. Let's look at the tables required to define this relationship:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Though `posts` does not contain a `country_id` column, the `hasManyThrough` relation provides access to a country's posts via `$country->posts`. To perform this query, Eloquent inspects the `country_id` on the intermediate `users` table. After finding the matching user IDs, they are used to query the `posts` table.

Now that we have examined the table structure for the relationship, let's define it on the `Country` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
     * Get all of the posts for the country.
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

The first argument passed to the `hasManyThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model, while the fourth argument is the name of the foreign key on the final model.

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');
    }
}
```

Polymorphic Relations

Table Structure

Polymorphic relations allow a model to belong to more than one other model on a single association. For example, imagine you want to store photos for your staff members and for your products. Using polymorphic relationships, you can use a single `photos` table for both of these scenarios. First, let's examine the table structure required to build this relationship:

```
staff
    id - integer
    name - string
```

```
products
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

Two important columns to note are the `imageable_id` and `imageable_type` columns on the `photos` table. The `imageable_id` column will contain the ID value of the owning staff or product, while the `imageable_type` column will contain the class name of the owning model. The `imageable_type` column is how the ORM determines which "type" of owning model to return when accessing the `imageable` relation.

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Photo extends Model
{
    /**
     * Get all of the owning imageable models.
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}

class Staff extends Model
{
    /**
     * Get all of the staff member's photos.
     */
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}

class Product extends Model
{
    /**
     * Get all of the product's photos.
     */
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
```

Retrieving Polymorphic Relations

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the photos for a staff member, we can simply use the `photos` dynamic property:

```
$staff = App\Staff::find(1);

foreach ($staff->photos as $photo) {
    //
}
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In our case, that is the `imageable` method on the `Photo` model. So, we will access that method as a dynamic property:

```
$photo = App\Photo::find(1);
```

```
$imageable = $photo->imageable;
```

The `imageable` relation on the `Photo` model will return either a `Staff` or `Product` instance, depending on which type of model owns the photo.

Custom Polymorphic Types

By default, Laravel will use the fully qualified class name to store the type of the related model. For instance, given the example above where a `Like` may belong to a `Post` or a `Comment`, the default `likable_type` would be either `App\Post` or `App\Comment`, respectively. However, you may wish to decouple your database from your application's internal structure. In that case, you may define a relationship "morph map" to instruct Eloquent to use the table name associated with each model instead of the class name:

```
Relation::morphMap([
    App\Post::class,
    App\Comment::class,
]);
```

Or, you may specify a custom string to associate with each model:

```
Relation::morphMap([
    'posts' => App\Post::class,
    'likes' => App\Like::class,
]);
```

You may register the `morphMap` in your `AppServiceProvider` or create a separate service provider if you wish.

Many To Many Polymorphic Relations

Table Structure

In addition to traditional polymorphic relations, you may also define "many-to-many" polymorphic relations. For example, a blog `Post` and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation allows you to have a single list of unique tags that are shared across blog posts and videos. First, let's examine the table structure:

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

Model Structure

Next, we're ready to define the relationships on the model. The `Post` and `Video` models will both have a `tags` method that calls the `morphToMany` method on the base Eloquent class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
```

```
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

Defining The Inverse Of The Relationship

Next, on the `Tag` model, you should define a method for each of its related models. So, for this example, we will define a `posts` method and a `videos` method:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the tags for a post, you can simply use the `tags` dynamic property:

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphedByMany`. In our case, that is the `posts` or `videos` methods on the `Tag` model. So, you will access those methods as dynamic properties:

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

Querying Relations

Since all types of Eloquent relationships are defined via functions, you may call those functions to obtain an instance of the relationship without actually executing the relationship queries. In addition, all types of Eloquent relationships also serve as query builders, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL against your database.

For example, imagine a blog system in which a `User` model has many associated `Post` models:

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

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

$user->posts()->where('active', 1)->get();
```

Note that you are able to use any of the query builder methods on the relationship!

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may simply access the relationship as if it were a property. For example, continuing to use our `User` and `Post` example models, we may access all of a user's posts like so:

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

foreach ($user->posts as $post) {
    //
}
```

Dynamic properties are "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use eager loading to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

Querying Relationship Existence

When accessing the records for a model, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the `has` method:

```
// Retrieve all posts that have at least one comment...
$posts = App\Post::has('comments')->get();
```

You may also specify an operator and count to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may also be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment and vote:

```
// Retrieve all posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to put "where" conditions on your `has` queries. These methods allow you to add customized constraints to a relationship constraint, such as checking the content of a comment:

```
// Retrieve all posts with at least one comment containing words like foo%
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Eager Loading

When accessing Eloquent relationships as properties, the relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the N + 1 query problem. To illustrate the N + 1 query problem, consider a `Book` model that is related to `Author`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Now, let's retrieve all books and their authors:

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, this loop would run 26 queries: 1 for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just 2 queries. When querying, you may specify which relationships should be eager loaded using the `with` method:

```
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships in a single operation. To do so, just pass additional arguments to the `with` method:

```
$books = App\Book::with('author', 'publisher')->get();
```

Nested Eager Loading

To eager load nested relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts in one Eloquent statement:

```
$books = App\Book::with('author.contacts')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship, but also specify additional query constraints for the eager loading query. Here's an example:

```
$users = App\User::with(['posts' => function ($query) {
```

```
    $query->where('title', 'like', '%first%');

}])->get();
```

In this example, Eloquent will only eager load posts that if the post's `title` column contains the word `first`. Of course, you may call other query builder to further customize the eager loading operation:

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');

}])->get();
```

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

If you need to set additional query constraints on the eager loading query, you may pass a `Closure` to the `load` method:

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

Inserting Related Models

The Save Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to insert a new `Comment` for a `Post` model. Instead of manually setting the `post_id` attribute on the `Comment`, you may insert the `Comment` directly from the relationship's `save` method:

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

Notice that we did not access the `comments` relationship as a dynamic property. Instead, we called the `comments` method to obtain an instance of the relationship. The `save` method will automatically add the appropriate `post_id` value to the new `Comment` model.

If you need to save multiple related models, you may use the `saveMany` method:

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

Save & Many To Many Relationships

When working with a many-to-many relationship, the `save` method accepts an array of additional intermediate table attributes as its second argument:

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

The Create Method

In addition to the `save` and `saveMany` methods, you may also use the `create` method, which accepts an array of

attributes, creates a model, and inserts it into the database. Again, the difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`:

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Before using the `create` method, be sure to review the documentation on attribute mass assignment.

Updating "Belongs To" Relationships

When updating a `belongsTo` relationship, you may use the `associate` method. This method will set the foreign key on the child model:

```
$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();
```

When removing a `belongsTo` relationship, you may use the `dissociate` method. This method will reset the foreign key as well as the relation on the child model:

```
$user->account()->dissociate();

$user->save();
```

Many To Many Relationships

Attaching / Detaching

When working with many-to-many relationships, Eloquent provides a few additional helper methods to make working with related models more convenient. For example, let's imagine a user can have many roles and a role can have many users. To attach a role to a user by inserting a record in the intermediate table that joins the models, use the `attach` method:

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

$user->roles()->attach($roleId);
```

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Of course, sometimes it may be necessary to remove a role from a user. To remove a many-to-many relationship record, use the `detach` method. The `detach` method will remove the appropriate record out of the intermediate table; however, both models will remain in the database:

```
// Detach a single role from the user...
$user->roles()->detach($roleId);

// Detach all roles from the user...
$user->roles()->detach();
```

For convenience, `attach` and `detach` also accept arrays of IDs as input:

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

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

Syncing For Convenience

You may also use the `sync` method to construct many-to-many associations. The `sync` method accepts an array

of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
```

You may also pass additional intermediate table values with the IDs:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

Touching Parent Timestamps

When a model `belongsTo` or `belongsToMany` another model, such as a `Comment` which belongs to a `Post`, it is sometimes helpful to update the parent's timestamp when the child model is updated. For example, when a `Comment` model is updated, you may want to automatically "touch" the `updated_at` timestamp of the owning `Post`. Eloquent makes it easy. Just add a `touches` property containing the names of the relationships to the child model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated as well:

```
$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

Eloquent ORM

Eloquent: Collections

Introduction

All multi-result sets returned by Eloquent are instances of the `Illuminate\Database\Eloquent\Collection` object, including results retrieved via the `get` method or accessed via a relationship. The Eloquent collection object extends the Laravel base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models.

Of course, all collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
$users = App\User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, let's remove all inactive models and gather the first name for each remaining user:

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})
->map(function ($user) {
    return $user->name;
});
```

> **Note:** While most Eloquent collection methods return a new instance of an Eloquent collection, the `pluck`, `keys`, `zip`, `collapse`, `flatten` and `flip` methods return a base collection instance. Likewise, if a `map` operation returns a collection that does not contain any Eloquent models, it will be automatically cast to a base collection.

Available Methods

The Base Collection

All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class:

all	has	reject
average	implode	reverse
avg	intersect	search
chunk	isEmpty	shift
collapse	keyBy	shuffle
contains	keys	slice
count	last	sort
diff	map	sortBy
each	max	sortByDesc
every	merge	splice
except	min	sum
filter	only	take
first	pluck	toArray
flatMap	pop	toJson

Custom Collections

If you need to use a custom `Collection` object with your own extension methods, you may override the `newCollection` method on your model:

```
<?php

namespace App;

use App\CustomCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Create a new Eloquent Collection instance.
     *
     * @param  array  $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

Once you have defined a `newCollection` method, you will receive an instance of your custom collection anytime Eloquent returns a `Collection` instance of that model. If you would like to use a custom collection for every model in your application, you should override the `newCollection` method on a base model class that is extended by all of your models.

Eloquent ORM

Eloquent: Mutators

Introduction

Accessors and mutators allow you to format Eloquent attributes when retrieving them from a model or setting their value. For example, you may want to use the Laravel encrypter to encrypt a value while it is stored in the database, and then automatically decrypt the attribute when you access it on an Eloquent model.

In addition to custom accessors and mutators, Eloquent can also automatically cast date fields to Carbon instances or even cast text fields to JSON.

Accessors & Mutators

Defining An Accessor

To define an accessor, create a `getFooAttribute` method on your model where `Foo` is the "camel" cased name of the column you wish to access. In this example, we'll define an accessor for the `first_name` attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of `first_name`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the user's first name.
     *
     * @param  string  $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the mutator, you may simply access the `first_name` attribute:

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

$firstName = $user->first_name;
```

Defining A Mutator

To define a mutator, define a `setFooAttribute` method on your model where `Foo` is the "camel" cased name of the column you wish to access. So, again, let's define a mutator for the `first_name` attribute. This mutator will be automatically called when we attempt to set the value of the `first_name` attribute on the model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
```

```
{
    /**
     * Set the user's first name.
     *
     * @param  string  $value
     * @return string
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

The mutator will receive the value that is being set on the attribute, allowing you to manipulate the value and set the manipulated value on the Eloquent model's internal `$attributes` property. So, for example, if we attempt to set the `first_name` attribute to `Sally`:

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

$user->first_name = 'Sally';
```

In this example, the `setFirstNameAttribute` function will be called with the value `Sally`. The mutator will then apply the `strtolower` function to the name and set its value in the internal `$attributes` array.

Date Mutators

By default, Eloquent will convert the `created_at` and `updated_at` columns to instances of Carbon, which provides an assortment of helpful methods, and extends the native PHP `DateTime` class.

You may customize which fields are automatically mutated, and even completely disable this mutation, by overriding the `$dates` property of your model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = ['created_at', 'updated_at', 'deleted_at'];
}
```

When a column is considered a date, you may set its value to a UNIX timestamp, date string (`Y-m-d`), date-time string, and of course a `DateTime` / `Carbon` instance, and the date's value will automatically be correctly stored in your database:

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

$user->deleted_at = Carbon::now();

$user->save();
```

As noted above, when retrieving attributes that are listed in your `$dates` property, they will automatically be cast to Carbon instances, allowing you to use any of Carbon's methods on your attributes:

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

return $user->deleted_at->getTimestamp();
```

By default, timestamps are formatted as `'Y-m-d H:i:s'`. If you need to customize the timestamp format, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database, as well as their format when the model is serialized to an array or JSON:

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Attribute Casting

The `$casts` property on your model provides a convenient method of converting attributes to common data types. The `$casts` property should be an array where the key is the name of the attribute being cast, while the value is the type you wish to cast to the column to. The supported cast types are: `integer`, `real`, `float`, `double`, `string`, `boolean`, `object`, `array`, `collection`, `date` and `datetime`.

For example, let's cast the `is_admin` attribute, which is stored in our database as an integer (`0` or `1`) to a boolean value:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be casted to native types.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

Now the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer:

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

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

Array Casting

The `array` cast type is particularly useful when working with columns that are stored as serialized JSON. For example, if your database has a `TEXT` field type that contains serialized JSON, adding the `array` cast to that attribute will automatically deserialize the attribute to a PHP array when you access it on your Eloquent model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be casted to native types.
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

Once the cast is defined, you may access the `options` attribute and it will automatically be deserialized from JSON into a PHP array. When you set the value of the `options` attribute, the given array will automatically be serialized back into JSON for storage:

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

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

Eloquent ORM

Eloquent: Serialization

Introduction

When building JSON APIs, you will often need to convert your models and relationships to arrays or JSON. Eloquent includes convenient methods for making these conversions, as well as controlling which attributes are included in your serializations.

Basic Usage

Converting A Model To An Array

To convert a model and its loaded relationships to an array, you may use the `toArray` method. This method is recursive, so all attributes and all relations (including the relations of relations) will be converted to arrays:

```
$user = App\User::with('roles')->first();

return $user->toArray();
```

You may also convert collections to arrays:

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

return $users->toArray();
```

Converting A Model To JSON

To convert a model to JSON, you may use the `toJson` method. Like `toArray`, the `toJson` method is recursive, so all attributes and relations will be converted to JSON:

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

return $user->toJson();
```

Alternatively, you may cast a model or collection to a string, which will automatically call the `toJson` method:

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

return (string) $user;
```

Since models and collections are converted to JSON when cast to a string, you can return Eloquent objects directly from your application's routes or controllers:

```
Route::get('users', function () {
    return App\User::all();
});
```

Hiding Attributes From JSON

Sometimes you may wish to limit the attributes, such as passwords, that are included in your model's array or JSON representation. To do so, add a `$hidden` property definition to your model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

Note: When hiding relationships, use the relationship's **method** name, not its dynamic property name.

Alternatively, you may use the `visible` property to define a white-list of attributes that should be included in your model's array and JSON representation:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be visible in arrays.
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

Appending Values To JSON

Occasionally, you may need to add array attributes that do not have a corresponding column in your database. To do so, first define an accessor for the value:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the administrator flag for the user.
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

Once you have created the accessor, add the attribute name to the `appends` property on the model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The accessors to append to the model's array form.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

Once the attribute has been added to the `appends` list, it will be included in both the model's array and JSON

forms. Attributes in the `appends` array will also respect the `visible` and `hidden` settings configured on the model.