Laravel 5.0 Documentation

Gani Utomo

Published with GitBook

Table of Contents

Introduction

A GitBook version of Laravel 5.0 Documentation

Prologue

Release Notes

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 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 IoC 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:

```
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:

```
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:

```
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 synchonrously (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 IoC 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 compliment the new default application structure, new Artisan generator commands have been added to the framework. See `php artisan list` for more details.

Configuration Cache

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

Symfony VarDumper

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

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

Laravel 4.2

The full change list for this release by running the `php artisan changes` command from a 4.2 installation, or by viewing the

change file on Github. These notes only cover the major enhancements and changes for the release.

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

PHP 5.4 Requirement

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

Laravel Forge

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

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

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

Laravel Homestead

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

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

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

Laravel Cashier

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

Daemon Queue Workers

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

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

Mail API Drivers

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

Soft Deleting Traits

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

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

Convenient Auth & Remindable Traits

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

"Simple Paginate"

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

Migration Confirmation

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

Laravel 4.1

Full Change List

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

New SSH Component

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

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

Boris In Tinker

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

Eloquent Improvements

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

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

Database Read / Write Connections

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

Queue Priority

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

Failed Queue Job Handling

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

Cache Tags

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

Flexible Password Reminders

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

Improved Routing Engine

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

Improved Session Engine

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

Doctrine DBAL

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

Upgrade Guide

Upgrading To 5.0 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 to your application.

To start, install a new Laravel 5 application into a fresh directory in your local environment. 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 IoC 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`. To fix this, add `"illuminate/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:

```
'Illuminate\Html\HtmlServiceProvider',
```

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

```
'Form'      => 'Illuminate\Html\FormFacade',
'Html'      => 'Illuminate\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()`.

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' => array(
    '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()`.

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 unit 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

Discussion regarding bugs, new features, and implementation of existing features takes place in the `#laravel-dev` IRC channel (Freenode). 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.

The `#laravel-dev` IRC channel is open to all. All are welcome to join the channel either to participate or simply observe the discussions!

Which Branch?

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

Minor features that are **fully 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 `#laravel-dev` IRC channel

(Freenode).

Security Vulnerabilities

If you discover a security vulnerability within Laravel, please send an e-mail to Taylor Otwell at taylorotwell@gmail.com. All security vulnerabilities will be promptly addressed.

Coding Style

Laravel follows the PSR-0 and PSR-1 coding standards. In addition to these standards, the following coding standards should be followed:

- The class namespace declaration must be on the same line as `<?php`.
- A class' opening `{` must be on the same line as the class name.
- Functions and control structures must use Allman style braces.
- Indent with tabs, align with spaces.

Setup

Installation

Install Composer

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

Install Laravel

Via Laravel Installer

First, download the Laravel installer using Composer.

```
composer global require "laravel/installer=~1.1"
```

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` would create a directory named `blog` containing a fresh Laravel installation with all dependencies installed. This method of installation is much faster than installing via Composer:

```
laravel new blog
```

Via Composer Create-Project

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

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

Server Requirements

The Laravel framework has a few system requirements:

- PHP >= 5.4
- Mcrypt PHP Extension
- OpenSSL PHP Extension
- Mbstring PHP Extension

As of PHP 5.5, some OS distributions may require you to manually install the PHP JSON extension. When using Ubuntu, this can be done via `apt-get install php5-json`.

Configuration

The first thing you should do after installing Laravel is set your application key to a random string. If you installed Laravel via Composer, this key has probably 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 the application key is not set, your user sessions and other encrypted data will not be secure!**

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.

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

> **Note:** You should never have the `app.debug` configuration option set to `true` for a production application.

Permissions

Laravel may require some permissions to be configured: folders within `storage` require write access by the web server.

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.

Configuration

Introduction

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

After Installation

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.

Other Configuration

Laravel needs very little 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 location.

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

> **Note:** You should never have the `app.debug` configuration option set to `true` for a production application.

Permissions

Laravel may require one set of permissions to be configured: folders within `storage` require write access by the web server.

Accessing Configuration Values

You may easily access your configuration values using the `Config` facade:

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

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

You may also use the `config` helper function:

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

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

You may access the current application environment via the `environment` method on the `Application` instance:

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

You may also pass arguments to the `environment` method to check if the environment matches a given value:

```
if ($app->environment('local'))
{
    // The environment is local
}

if ($app->environment('local', 'staging'))
{
    // The environment is either local OR staging...
}
```

To obtain an instance of the application, resolve the `Illuminate\Contracts\Foundation\Application` contract via the service container. Of course, if you are within a service provider, the application instance is available via the `$this->app` instance variable.

An application instance may also be accessed via the `app` helper of the `App` facade:

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

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

Configuration Caching

To give your application a little speed boost, you may 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 `config:cache` command as part of your deployment routine.

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.

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.

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 5.6, MySQL, Postgres, Redis, Memcached, 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.

Homestead is currently built and tested using Vagrant 1.6.

Included Software

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- Postgres
- Node (With Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy
- Fabric + HipChat Extension

Installation & Setup

Installing VirtualBox & Vagrant

Before launching your Homestead environment, you must install VirtualBox and Vagrant. Both of these software packages provide easy-to-use visual installers for all popular operating systems.

Adding The Vagrant Box

Once VirtualBox 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
```

Installing Homestead

Manually Via Git (No Local PHP)

Alternatively, if you do not want to install PHP on your local machine, you may install Homestead manually 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 (and PHP) projects:

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

Once you have installed the Homestead CLI tool, run the `bash init.sh` command to create the `Homestead.yaml` configuration file:

```
bash init.sh
```

The `Homestead.yaml` file will be placed in your `~/.homestead` directory.

With Composer + PHP Tool

Once the box has been added to your Vagrant installation, you are ready to install the Homestead CLI tool using the Composer `global` command:

```
composer global require "laravel/homestead=~2.0"
```

Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `homestead` executable is found when you run the `homestead` command in your terminal.

Once you have installed the Homestead CLI tool, run the `init` command to create the `Homestead.yaml` configuration file:

```
homestead init
```

The `Homestead.yaml` file will be placed in the `~/.homestead` directory. If you're using a Mac or Linux system, you may edit `Homestead.yaml` file by running the `homestead edit` command in your terminal:

```
homestead edit
```

Set Your SSH Key

Next, you should edit the `Homestead.yaml` file. In this file, you can configure the path to your public SSH key, as well as the folders you wish to be shared between your main machine and the Homestead virtual machine.

Don't have an SSH key? On Mac and Linux, you can generally create an SSH key pair using the following command:

```
ssh-keygen -t rsa -C "you@homestead"
```

On Windows, you may install Git and use the `Git Bash` shell included with Git to issue the command above. Alternatively, you may use PuTTY and PuTTYgen.

Once you have created a SSH key, specify the key's path in the `authorize` property of your `Homestead.yaml` file.

Configure Your 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!

Configure Your 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!

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
```

Bash Aliases

To add Bash aliases to your Homestead box, simply add to the `aliases` file in the root of the `~/.homestead` directory.

Launch 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 configure your shared folders and Nginx sites automatically! To destroy the machine, you may use the `vagrant destroy --force` command.

Don't forget to add the "domains" for your Nginx sites to the `hosts` file on your machine! The `hosts` file will redirect your requests for the local domains into your Homestead environment. 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 you set in your `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
```

To learn how to connect to your databases, read on!

Daily Usage

Connecting Via SSH

To connect to your Homestead environment via SSH, issue the `vagrant ssh` command from your Homestead directory.

Since you will probably need to SSH into your Homestead machine frequently, consider creating an "alias" on your host machine:

```
alias vm="ssh vagrant@127.0.0.1 -p 2222"
```

Once you create this alias, you can simply use the "vm" command to SSH into your Homestead machine from anywhere on your system.

Connecting To Your Databases

A `homestead` database is configured for both MySQL and Postgres out of the box. For even more convenience, Laravel's `local` database configuration is set to use this database by default.

To connect to your MySQL or Postgres database from your main 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 main 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. There are two ways to do this: First, you may simply add the sites to your `Homestead.yaml` file and then run `vagrant provision`.

Alternatively, you may use the `serve` script that is available on your Homestead environment. To use the `serve` script, SSH into your Homestead environment and run the following command:

```
serve domain.app /home/vagrant/Code/path/to/public/directory
```

> **Note:** After running the `serve` command, do not forget to add the new site to the `hosts` file on your main machine!

Ports

The following ports are forwarded to your Homestead environment:

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

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`:

Basic GET Route

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

Other Basic Routes Route

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

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

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

Registering A Route For Multiple Verbs

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

Registering A Route That Responds To Any HTTP Verb

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

Often, you will need to generate URLs to your routes, you may do so using the `url` helper:

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

CSRF Protection

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.

Insert The CSRF Token Into A Form

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

Of course, using the Blade templating engine:

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

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

In addition to looking for the CSRF token as a "POST" parameter, the middleware will also check for the `X-XSRF-TOKEN` request header, which is commonly used by JavaScript frameworks.

Method Spoofing

HTML forms do not support `PUT` or `DELETE` actions. So, when defining `PUT` 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. For example:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
</form>
```

Route Parameters

Of course, you can capture segments of the request URI within your route:

Basic Route Parameter

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

Optional Route Parameters

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

Optional Route Parameters With Default Value

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

Regular Expression Parameter Constraints

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

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

Passing An Array Of Constraints

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

Defining Global Patterns

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 `before` method of your `RouteServiceProvider`:

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

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

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

Accessing A Route Parameter Value

If you need to access a route parameter value outside of a route, use the `input` method:

```
if ($route->input('id') == 1)
{
    //
}
```

You may also access the current route parameters via the `Illuminate\Http\Request` instance. The request instance for the current request may be accessed via the `Request` facade, or by type-hinting the `Illuminate\Http\Request` where dependencies are injected:

```
use Illuminate\Http\Request;

Route::get('user/{id}', function(Request $request, $id)
{
    if ($request->route('id'))
    {
        //
    }
});
```

Named Routes

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

```
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'
]);
```

Now, you may use the route's name when generating URLs or redirects:

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

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

The `currentRouteName` method returns the name of the route handling the current request:

```
$name = Route::currentRouteName();
```

Route Groups

Sometimes you may need to apply filters to a group of routes. Instead of specifying the filter on each route, you may use a route group:

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

    Route::get('user/profile', function()
    {
        // Has Auth Filter
```

```
    });
});
```

You may use the `namespace` parameter within your `group` array to specify the namespace for all controllers within the group:

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

> **Note:** By default, the `RouteServiceProvider` includes your `routes.php` file within a namespace group, allowing you to register controller routes without specifying the full namespace.

Sub-Domain Routing

Laravel routes also handle wildcard sub-domains, and will pass your wildcard parameters from the domain:

Registering Sub-Domain Routes

```
Route::group(['domain' => '{account}.myapp.com'], function()
{

    Route::get('user/{id}', function($account, $id)
    {
        //
    });

});
```

Route Prefixing

A group of routes may be prefixed by using the `prefix` option in the attributes array of a group:

```
Route::group(['prefix' => 'admin'], function()
{

    Route::get('user', function()
    {
        //
    });

});
```

Route Model Binding

Laravel 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:

```
Route::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 error will be thrown.

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

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

If you wish to use your own resolution logic, you should use the `Router::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:

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

Throwing 404 Errors

There are two ways to manually trigger a 404 error from a route. First, you may use the `abort` helper:

```
abort(404);
```

The `abort` helper simply throws a `Symfony\Component\HttpFoundation\Exception\HttpException` with the specified status code.

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.

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, 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;

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 `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.

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.

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()
{
    //
}]);
```

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, you may define the middleware as "terminable".

```
use Illuminate\Contracts\Routing\TerminableMiddleware;

class StartSession implements TerminableMiddleware {

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

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

}
```

As you can see, in addition to defining a `handle` method, `TerminableMiddleware` define a `terminate` method. This method receives 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.

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:

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

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');
```

> **Note:** All controllers should extend the base controller class.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace, only 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

To generate a URL to a controller action, use the `action` helper method:

```
$url = action('App\Http\Controllers\FooController@method');
```

If you wish to generate a URL to a controller action while using only the portion of the class name relative to your controller namespace, register the root controller namespace with the URL generator:

```
URL::setRootControllerNamespace('App\Http\Controllers');

$url = action('FooController@method');
```

You may access the name of the controller action being run using the `currentRouteAction` method:

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

Controller Middleware

Middleware may be specified on controller routes like so:

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

Additionally, you may specify middleware within your controller's constructor:

```
class UserController extends Controller {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }

}
```

Implicit Controllers

Laravel allows you to easily define a single route to handle every action in a controller. First, define the route using the `Route::controller` method:

```
Route::controller('users', 'UserController');
```

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. Next, just add methods to your controller, prefixed with the HTTP verb they respond to:

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

The `index` methods will respond to the root URI handled by the controller, which, in this case, is `users`.

If your controller action contains multiple words, you may access the action using "dash" syntax in the URI. For example, the following controller action on our `UserController` would respond to the `users/admin-profile` URI:

```
public function getAdminProfile() {}
```

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
```

Next, we 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	/resource	index	resource.index

GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{resource}	show	resource.show
GET	/resource/{resource}/edit	edit	resource.edit
PUT/PATCH	/resource/{resource}	update	resource.update
DELETE	/resource/{resource}	destroy	resource.destroy

Customizing Resource Routes

Additionally, you may specify only 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']]);
```

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']]);
```

Handling Nested Resource Controllers

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}`.

```
class PhotoCommentController extends Controller {

    /**
     * Show the specified photo comment.
     *
     * @param  int  $photoId
     * @param  int  $commentId
     * @return Response
     */
    public function show($photoId, $commentId)
    {
        //
    }

}
```

Adding Additional Routes To 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`:

```
Route::get('photos/popular');
```

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

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:

```
<?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 methods. For example, let's type-hint the `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:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

> **Note:** Method injection is fully compatible with model binding. The container will intelligently determine which arguments are model bound and which arguments should be injected.

Route Caching

If your application is exclusively using controller routes, you may take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it take 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
```

HTTP Requests

Obtaining A Request Instance

Via Facade

The `Request` facade will grant you access to the current request that is bound in the container. For example:

```
$name = Request::input('name');
```

Remember, if you are in a namespace, you will have to import the `Request` facade using a `use Request;` statement at the top of your class file.

Via Dependency Injection

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the 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:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
```

```
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

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');
```

Retrieving A Default Value If The Input Value Is Absent

```
$name = Request::input('name', 'Sally');
```

Determining If An Input Value Is Present

```
if (Request::has('name'))
{
    //
}
```

Getting All Input For The Request

```
$input = Request::all();
```

Getting Only Some Of The Request Input

```
$input = Request::only('username', 'password');

$input = Request::except('credit_card');
```

When working on forms with "array" inputs, you may use dot notation to access the arrays:

```
$input = Request::input('products.0.name');
```

Old Input

Laravel also allows you to keep input from one request during the next request. For example, you may need to re-populate a form after checking it for validation errors.

Flashing Input To The Session

The `flash` method will flash the current input to the session so that it is available during the user's next request to the application:

```
Request::flash();
```

Flashing Only Some Input To The Session

```
Request::flashOnly('username', 'email');

Request::flashExcept('password');
```

Flash & 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.

```
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.

```
$username = Request::old('username');
```

If you are displaying old input within a Blade template, it is more convenient to use the `old` helper:

```
{{ old('username') }}
```

Cookies

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.

Retrieving A Cookie Value

```
$value = Request::cookie('name');
```

Attaching A New Cookie To A Response

The `cookie` helper serves as a simple factory for generating new `Symfony\Component\HttpFoundation\Cookie` instances. The cookies may be attached to a `Response` instance using the `withCookie` method:

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));
```

Creating A Cookie That Lasts Forever*

By "forever", we really mean five years.

```
$response->withCookie(cookie()->forever('name', 'value'));
```

Files

Retrieving An Uploaded File

```
$file = Request::file('photo');
```

Determining If A File Was Uploaded

```
if (Request::hasFile('photo'))
{
    //
}
```

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.

Determining If An Uploaded File Is Valid

```
if (Request::file('photo')->isValid())
{
    //
}
```

Moving An Uploaded File

```
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.

Other Request Information

The `Request` class provides many methods for examining the HTTP request for your application and extends the `Symfony\Component\HttpFoundation\Request` class. Here are some of the highlights.

Retrieving The Request URI

```
$uri = Request::path();
```

Retrieving The Request Method

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

Determining If The Request Path Matches A Pattern

```
if (Request::is('admin/*'))
{
    //
}
```

Get The Current Request URL

```
$url = Request::url();
```

HTTP Responses

Basic Responses

Returning Strings From Routes

The most basic response from a Laravel route is a string:

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

Creating Custom Responses

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 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;

return (new Response($content, $status))
              ->header('Content-Type', $value);
```

For convenience, you may also use the `response` helper:

```
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.

Sending A View In A Response

If you need access to the `Response` class methods, but want to return a view as the response content, you may use the `view` method for convenience:

```
return response()->view('hello')->header('Content-Type', $type);
```

Attaching Cookies To Responses

```
return response($content)->withCookie(cookie('name', 'value'));
```

Method Chaining

Keep in mind that most `Response` methods are chainable, allowing for the fluent building of responses:

```
return response()->view('hello')->header('Content-Type', $type)
                 ->withCookie(cookie('name', 'value'));
```

Redirects

Redirect responses are typically instances of the `Illuminate\Http\RedirectResponse` class, and contain the proper headers needed to redirect the user to another URL.

Returning A Redirect

There are several ways to generate a `RedirectResponse` instance. The simplest method is to use the `redirect` helper method. When testing, it is not common to mock the creation of a redirect response, so using the helper method is almost always acceptable:

```
return redirect('user/login');
```

Returning A Redirect With Flash 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:

```
return redirect('user/login')->with('message', 'Login Failed');
```

Redirecting To The Previous URL

You may wish to redirect the user to their previous location, for example, after a form submission. You can do so by using the `back` method:

```
return redirect()->back();

return redirect()->back()->withInput();
```

Returning A Redirect To A Named Route

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');
```

Returning A Redirect To A Named Route With Parameters

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]);
```

Returning A Redirect To A Named Route Using Named Parameters

```
// For a route with the following URI: profile/{user}

return redirect()->route('profile', ['user' => 1]);
```

Returning A Redirect To A Controller Action

Similarly to generating `RedirectResponse` instances to named routes, you may also generate redirects to controller actions:

```
return redirect()->action('App\Http\Controllers\HomeController@index');
```

> **Note:** You do not need to specify the full namespace to the controller if you have registered a root controller namespace via `URL::setRootControllerNamespace`.

Returning A Redirect To A Controller Action With Parameters

```
return redirect()->action('App\Http\Controllers\UserController@profile', [1]);
```

Returning A Redirect To A Controller Action Using Named Parameters

```
return redirect()->action('App\Http\Controllers\UserController@profile', ['user' => 1]);
```

Other Responses

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.

Creating A JSON Response

The `json` method will automatically set the `Content-Type` header to `application/json`:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

Creating A JSONP Response

```
return response()->json(['name' => 'Abigail', 'state' => 'CA'])
                 ->setCallback($request->input('callback'));
```

Creating A File Download Response

```
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.

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 Response;
use Illuminate\Support\ServiceProvider;

class ResponseMacroServiceProvider extends ServiceProvider {

    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function($value)
        {
            return Response::make(strtoupper($value));
        });
    }

}
```

The `macro` function accepts a name as its first argument, and a Closure as its second. The macro's Closure will be executed when calling the macro name from a `ResponseFactory` implementation or the `response` helper:

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

Views

Basic Usage

Views contain the HTML served by your application, and serve as a convenient method of separating your controller and domain logic from your presentation logic. Views are stored in the `resources/views` directory.

A simple view looks like this:

```
<!-- View stored in resources/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

The view may be returned to the browser 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.

Of course, views may also be nested within sub-directories of the `resources/views` directory. For example, if your view is stored at `resources/views/admin/profile.php`, it should be returned like so:

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

Passing Data To Views

```
// Using conventional approach
$view = view('greeting')->with('name', 'Victoria');

// Using Magic Methods
$view = view('greeting')->withName('Victoria');
```

In the example above, the variable `$name` is made accessible to the view and contains `Victoria`.

If you wish, you may pass an array of data as the second parameter to the `view` helper:

```
$view = view('greetings', $data);
```

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 have several options: the `view` helper, the `Illuminate\Contracts\View\Factory` contract, or a wildcard view composer.

For example, using the `view` helper:

```
view()->share('data', [1, 2, 3]);
```

You may also use the `View` facade:

```
View::share('data', [1, 2, 3]);
```

Typically, you would place calls to the `share` method 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.

> **Note:** When the `view` helper is called without arguments, it returns an implementation of the `Illuminate\Contracts\View\Factory` contract.

Determining If A View Exists

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

```
if (view()->exists('emails.customer'))
{
    //
}
```

Returning A View From A File Path

If you wish, you may generate a view from a fully-qualified file path:

```
return view()->file($pathToFile, $data);
```

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 organizes that logic into a single location.

Defining A View Composer

Let's organize our view composers within a service provider. We'll use the `View` facade to access the underlying `Illuminate\Contracts\View\Factory` contract implementation:

```
<?php namespace App\Providers;

use View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider {

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
```

```
        // Using class based composers...
        View::composer('profile', 'App\Http\ViewComposers\ProfileComposer');

        // Using Closure based composers...
        View::composer('dashboard', function()
        {

        });
    }

}
```

> **Note:** 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\Composers` directory.

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\Composers;

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.

Wildcard View Composers

The `composer` method accepts the `*` character as a wildcard, so you may attach a composer to all views like so:

```
View::composer('*', function()
{
    //
```

```
});
```

Attaching A Composer To Multiple Views

You may also attach a view composer to multiple views at once:

```
View::composer(['profile', 'dashboard'], 'App\Http\ViewComposers\MyViewComposer');
```

Defining Multiple Composers

You may use the `composers` method to register a group of composers at the same time:

```
View::composers([
    'App\Http\ViewComposers\AdminComposer' => ['admin.index', 'admin.profile'],
    'App\Http\ViewComposers\UserComposer' => 'user',
    'App\Http\ViewComposers\ProductComposer' => 'product'
]);
```

View Creators

View **creators** work almost exactly like view composers; however, they are fired immediately when the view is instantiated. To register a view creator, use the `creator` method:

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

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, filters, 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.

Basic Provider Example

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

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($app['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, don't worry, we'll cover that soon.

This class is namespaced under `App\Providers` since that is the default location for service providers in Laravel. However, you are free to change this as you wish. Your service providers may be placed anywhere that Composer can autoload them.

The Boot Method

So, what if we need to register an event listener 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 Event;
use Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider {

    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Event::listen('SomeEvent', 'SomeEventHandler');
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

}
```

We are able to type-hint dependencies for our `boot` method. The service container will automatically inject any dependencies you need:

```
use Illuminate\Contracts\Events\Dispatcher;

public function boot(Dispatcher $events)
{
    $events->listen('SomeEvent', 'SomeEventHandler');
}
```

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',
],
```

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.

Service Container

Introduction

The Laravel service container is a powerful tool for managing class dependencies. Dependency injection is a fancy word 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\Handlers\Commands;

use App\User;
use App\Commands\PurchasePodcast;
use Illuminate\Contracts\Mail\Mailer;

class PurchasePodcastHandler {

    /**
     * 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.
     *
     * @param  PurchasePodcastCommand  $command
     * @return void
     */
    public function handle(PurchasePodcastCommand $command)
    {
        //
    }

}
```

In this example, the `PurchasePodcast` command handler 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.

Basic Usage

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, if you need an instance of the container elsewhere in your application, such as a factory, you may type-hint the `Illuminate\Contracts\Container\Container` contract and an instance of the container will be injected for you. Alternatively, you may use the `App` facade to access the container.

Registering A Basic Resolver

Within a service provider, you always have access to the container via the `$this->app` instance variable.

There are several ways the service container can register dependencies, including Closure callbacks and binding interfaces to implementations. First, we'll explore Closure callbacks. A Closure resolver is registered in the container with a key (typically the class name) and a Closure that returns some value:

```
$this->app->bind('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

Registering A Singleton

Sometimes, you may wish to bind something into the container that should only be resolved once, and the same instance should be returned on subsequent calls into the container:

```
$this->app->singleton('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

Binding An Existing Instance Into The Container

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);
```

Resolving

There are several ways to resolve something out of the container. First, you may use the `make` method:

```
$fooBar = $this->app->make('FooBar');
```

Secondly, you may use "array access" on the container, 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, filters, and more. The container will automatically inject the dependencies:

```
<?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)
    {
        //
    }

}
```

Binding Interfaces To Implementations

Injecting Concrete Dependencies

A very powerful features of the service container is its ability to bind an interface to a given implementation. For example, perhaps our application integrates with the Pusher web service for sending and receiving real-time events. If we are using Pusher's PHP SDK, we could inject an instance of the Pusher client into a class:

```
<?php namespace App\Handlers\Commands;

use App\Commands\CreateOrder;
use Pusher\Client as PusherClient;

class CreateOrderHandler {

    /**
     * The Pusher SDK client instance.
     */
    protected $pusher;

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

    /**
     * Execute the given command.
```

```
     *
     * @param  CreateOrder  $command
     * @return void
     */
    public function execute(CreateOrder $command)
    {
        //
    }

}
```

In this example, it is good that we are injecting the class dependencies; however, we are tightly coupled to the Pusher SDK. If the Pusher SDK methods change or we decide to switch to a new event service entirely, we will need to change our `CreateOrderHandler` code.

Program To An Interface

In order to "insulate" the `CreateOrderHandler` against changes to event pushing, we could define an `EventPusher` interface and a `PusherEventPusher` implementation:

```
<?php namespace App\Contracts;

interface EventPusher {

    /**
     * Push a new event to all clients.
     *
     * @param  string  $event
     * @param  array  $data
     * @return void
     */
    public function push($event, array $data);

}
```

Once we have coded our `PusherEventPusher` implementation of this interface, we can register it with the service container like so:

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

This tells the container that it should inject the `PusherEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in our constructor:

```
    /**
     * Create a new order handler 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 definining this behavior:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
```

```
            ->needs('App\Contracts\EventPusher')
            ->give('App\Services\PubNubEventPusher');
```

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'));
});
```

Practical Applications

Laravel provides several opportunities to use the service container to increase the flexibility and testability of your application. One primary example is when resolving controllers. All controllers are resolved through the service container, meaning you can type-hint dependencies in a controller constructor, and they will automatically be injected.

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

use Illuminate\Routing\Controller;
use App\Repositories\OrderRepository;

class OrdersController extends Controller {

    /**
     * The order repository instance.
     */
    protected $orders;

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

    /**
     * Show all of the orders.
     *
     * @return Response
     */
    public function index()
    {
        $all = $this->orders->all();
```

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

}
```

In this example, the `OrderRepository` class will automatically be injected into the controller. This means that a "mock" `OrderRepository` may be bound into the container when unit testing, allowing for painless stubbing of database layer interaction.

Other Examples Of Container Usage

Of course, as mentioned above, controllers are not the only classes Laravel resolves via the service container. You may also type-hint dependencies on route Closures, filters, queue jobs, event listeners, and more. For examples of using the service container in these contexts, please refer to their documentation.

Container Events

Registering A Resolving Listener

The 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(function(FooBar $fooBar, $app)
{
    // Called when container resolves objects of type "FooBar"...
});
```

The object being resolved will be passed to the callback.

Contracts

Introduction

Laravel's Contracts are a set of interfaces that define the core services provided by the framework. For example, a `Queue` contract defines the methods needed for queueing jobs, while the `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 other package developers.

Why Contracts?

You may have several questions regarding contracts. Why use interfaces at all? Isn't using interfaces more complicated?

Let's distill the reasons for using interfaces to the following headings: loose coupling and simplicity.

Loose Coupling

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

```
<?php namespace App\Orders;

class Repository {

    /**
     * The cache.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param  \Package\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 {

    /**
     * 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	Laravel 4.x Facade
Illuminate\Contracts\Auth\Guard	Auth
Illuminate\Contracts\Auth\PasswordBroker	Password
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, filters, queue 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 handler:

```
<?php namespace App\Handlers\Events;

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 the documentation.

Facades

Introduction

Facades provide a "static" interface to classes that are available in the application's IoC 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 IoC container, providing the benefit of a terse, expressive syntax while maintaining more testability and flexibility than traditional static methods.

Occasionally, You may wish to create your own facades for your applications and packages, so let's explore the concept, development and usage of these classes.

> **Note:** Before digging into facades, it is strongly recommended that you become very familiar with the Laravel IoC container.

Explanation

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 `Facade` class.

Your 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.

So, when you make a facade call like `Cache::get`, Laravel resolves the Cache manager class out of the IoC container and calls the `get` method on the class. In technical terms, Laravel Facades are a convenient syntax for using the Laravel IoC container as a service locator.

Practical Usage

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.

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

However, 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'; }
```

```
}
```

The Cache class extends the base `Facade` class and defines a method `getFacadeAccessor()`. Remember, this method's job is to return the name of an IoC binding.

When a user references any static method on the `Cache` facade, Laravel resolves the `cache` binding from the IoC container and runs the requested method (in this case, `get`) against that object.

So, our `Cache::get` call could be re-written like so:

```
$value = $app->make('cache')->get('key');
```

Importing Facades

Remember, if you are using a facade when a controller that is namespaced, you will need to import the facade class into the namespace. All facades live in the global namespace:

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

use Cache;

class PhotosController extends Controller {

    /**
     * Get all of the application photos.
     *
     * @return Response
     */
    public function index()
    {
        $photos = Cache::get('photos');

        //
    }

}
```

Creating Facades

Creating a facade for your own application or package is simple. You only need 3 things:

- An IoC binding.
- A facade class.
- A facade alias configuration.

Let's look at an example. Here, we have a class defined as `PaymentGateway\Payment`.

```
namespace PaymentGateway;

class Payment {

    public function process()
    {
        //
    }

}
```

We need to be able to resolve this class from the IoC container. So, let's add a binding to a service provider:

```
App::bind('payment', function()
{
    return new \PaymentGateway\Payment;
});
```

A great place to register this binding would be to create a new service provider named `PaymentServiceProvider`, and add this binding to the `register` method. You can then configure Laravel to load your service provider from the `config/app.php` configuration file.

Next, we can create our own facade class:

```
use Illuminate\Support\Facades\Facade;

class Payment extends Facade {

    protected static function getFacadeAccessor() { return 'payment'; }

}
```

Finally, if we wish, we can add an alias for our facade to the `aliases` array in the `config/app.php` configuration file. Now, we can call the `process` method on an instance of the `Payment` class.

```
Payment::process();
```

A Note On Auto-Loading Aliases

Classes in the `aliases` array are not available in some instances because PHP will not attempt to autoload undefined type-hinted classes. If `\ServiceWrapper\ApiTimeoutException` is aliased to `ApiTimeoutException`, a `catch(ApiTimeoutException $e)` outside of the namespace `\ServiceWrapper` will never catch the exception, even if one is thrown. A similar problem is found in classes which have type hints to aliased classes. The only workaround is to forego aliasing and `use` the classes you wish to type hint at the top of each file which requires them.

Mocking Facades

Unit testing is an important aspect of why facades work the way that they do. In fact, testability is the primary reason for facades to even exist. For more information, check out the mocking facades section of the documentation.

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 IoC binding key is also included where applicable.

Facade	Class	IoC Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Console\Application	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
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`
Form	Illuminate\Html\FormBuilder	`form`
Hash	Illuminate\Hashing\HasherInterface	`hash`
HTML	Illuminate\Html\HtmlBuilder	`html`
Input	Illuminate\Http\Request	`request`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Paginator	Illuminate\Pagination\Factory	`paginator`
Paginator (Instance)	Illuminate\Pagination\Paginator	
Password	Illuminate\Auth\Passwords\PasswordBroker	`auth.reminder`
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\Support\Facades\Response	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	
SSH	Illuminate\Remote\RemoteManager	`remote`
SSH (Instance)	Illuminate\Remote\Connection	
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	

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.

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.

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.

The `config` directory, as the name implies, contains all of your application's configuration files.

The `database` folder contains your database migration and seeds.

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 "language" files.

The `storage` directory contains compiled Blade templates, file based sessions, file caches, and other files generated by the framework.

The `tests` directory contains your automated tests.

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, filters, and requests.

The `Commands` directory, of course, houses the commands for your application. Commands represent jobs that can be queued by your application, as well as tasks that you can run synchronously within the current request lifecycle.

The `Events` directory, as you might expect, houses event classes. Of course, using classes to represent events is not

required; however, if you choose to use them, this directory is the default location they will be created by the Artisan command line.

The `Handlers` directory contains the handler classes for both commands and events. Handlers receive a command or event and perform logic in response to that command or event being fired.

The `Services` directory contains various "helper" services your application needs to function. For example, the `Registrar` service included with Laravel is responsible for validating and creating new users of your application. Other examples might be services to interact with external APIs, metrics systems, or even services that aggregate data from your own application.

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
```

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.

By default, Laravel includes an `App\User` model in your `app` directory. This model may be used with the default Eloquent authentication driver.

Remember: when building the database schema for this model, make the password column at least 60 characters. Also, before getting started, make sure 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. Of course, Laravel 5 ships migrations for these columns out of the box!

If your application is not using Eloquent, you may use the `database` authentication driver which uses the Laravel query builder.

Authenticating Users

Laravel ships with two authentication related controllers out of the box. The `AuthController` handles new user registration and "logging in", 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. The views that these controllers render are located in the `resources/views/auth` directory. You are free to customize these views however you wish.

The User Registrar

To modify the form fields that are required when a new user registers with your application, you may modify the `App\Services\Registrar` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `Registrar` contains the validation rules for new users of the application, while the `create` method of the `Registrar` is responsible for creating new `User` records in your database. You are free to modify each of these methods as you wish. The `Registrar` is called by the `AuthController` via the methods contained in the `AuthenticatesAndRegistersUsers` trait.

Manual Authentication

If you choose not to use the provided `AuthController` implementation, you will need to manage the authentication of your users using the Laravel authentication classes directly. Don't worry, it's still a cinch! First, 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]))
        {
            return redirect()->intended('dashboard');
        }
    }

}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The `password` value will be hashed. The other 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, a new authenticated session will be started for the user.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

> **Note:** In this example, `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.

The `intended` redirect function 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.

Authenticating A User With Conditions

You also may add extra conditions to the authentication query:

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1]))
{
    // The user is active, not suspended, and exists.
}
```

Determining If A User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method:

```
if (Auth::check())
{
    // The user is logged in...
}
```

Authenticating A User And "Remembering" Them

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())
{
    //
}
```

Authenticating Users By ID

To log a user into the application by their ID, use the `loginUsingId` method:

```
Auth::loginUsingId(1);
```

Validating User Credentials Without Login

The `validate` method allows you to validate a user's credentials without actually logging them into the application:

```
if (Auth::validate($credentials))
{
    //
}
```

Logging A User In For A Single Request

You may also use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized:

```
if (Auth::once($credentials))
{
    //
}
```

Manually Logging In A User

If you need to log an existing user instance into your application, you may call the `login` method with the user instance:

```
Auth::login($user);
```

This is equivalent to logging in a user via credentials using the `attempt` method.

Logging A User Out Of The Application

```
Auth::logout();
```

Of course, if you are using the built-in Laravel authentication controllers, a controller method that handles logging users out of the application is provided out of the box.

Authentication Events

When the `attempt` method is called, the `auth.attempt` event will be fired. If the authentication attempt is successful and the user is logged in, the `auth.login` event will be fired as well.

Retrieving The Authenticated User

Once a user is authenticated, there are several ways to obtain an instance of the User.

First, you may access the user from the `Auth` facade:

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

use Illuminate\Routing\Controller;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile()
    {
        if (Auth::user())
        {
            // Auth::user() returns an instance of the authenticated user...
        }
    }

}
```

Second, 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.
     *
     * @return Response
     */
    public function updateProfile(Request $request)
    {
        if ($request->user())
        {
            // $request->user() returns an instance of the authenticated user...
        }
    }

}
```

Thirdly, you may type-hint the `Illuminate\Contracts\Auth\Authenticatable` contract. This type-hint may be added to a controller constructor, controller method, or any other constructor of a class resolved by the service container:

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

use Illuminate\Routing\Controller;
use Illuminate\Contracts\Auth\Authenticatable;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
```

```
     *
     * @return Response
     */
    public function updateProfile(Authenticatable $user)
    {
        // $user is an instance of the authenticated user...
    }

}
```

Protecting Routes

Route middleware can be used to allow only authenticated users to access a given route. Laravel provides the `auth` middleware by default, and it is defined in `app\Http\Middleware\Authenticate.php`. All you need to do is attach it to a route definition:

```
// With A Route Closure...

Route::get('profile', ['middleware' => 'auth', function()
{
    // Only authenticated users may enter...
}]);

// With A Controller...

Route::get('profile', ['middleware' => 'auth', 'uses' => 'ProfileController@show']);
```

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:

Protecting A Route With HTTP Basic

```
Route::get('profile', ['middleware' => 'auth.basic', function()
{
    // Only authenticated users may enter...
}]);
```

By default, the `basic` middleware will use the `email` column on the user record as the "username".

Setting Up A Stateless HTTP Basic Filter

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:

```
public function handle($request, Closure $next)
{
    return Auth::onceBasic() ?: $next($request);
}
```

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}]
```

Password Reminders & Reset

Model & Table

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 `User` model implements the `Illuminate\Contracts\Auth\CanResetPassword` contract. Of course, the `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 Reminder 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
```

Password Reminder Controller

Laravel also includes an `Auth\PasswordController` that contains the logic necessary to reset user passwords. We've even provided views to get you started! The views are located in the `resources/views/auth` directory. You are free to modify these views as you wish to suit your own application's design.

Your user will receive an e-mail with a link that points to the `getReset` method of the `PasswordController`. This method will render the password reset form and allow users to reset their passwords. After the password is reset, the user will automatically be logged into the application and redirected to `/home`. You can customize the post-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 of 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, Google, and GitHub.**

To get started with Socialite, include the package in your `composer.json` file:

```
"laravel/socialite": "~2.0"
```

Next, register the `Laravel\Socialite\SocialiteServiceProvider` in your `config/app.php` configuration file. You may also register a facade:

```
'Socialize' => 'Laravel\Socialite\Facades\Socialite',
```

You will 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`, `google`, or `github`, 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',
],
```

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. Here's an example using the `Socialize` facade:

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

public function handleProviderCallback()
{
    $user = Socialize::with('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:

```
return Socialize::with('github')->scopes(['scope1', 'scope2'])->redirect();
```

Once you have a user instance, you can grab a few more details about the user:

Retrieving User Details

```
$user = Socialize::with('github')->user();

// OAuth Two Providers
$token = $user->token;

// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

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:

```
"laravel/cashier": "~3.0"
```

Service Provider

Next, register the `Laravel\Cashier\CashierServiceProvider` 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 use `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 Eloquent implements BillableContract {

    use Billable;

    protected $dates = ['trial_ends_at', 'subscription_ends_at'];

}
```

Stripe Key

Finally, set your Stripe key in one of your bootstrap files or service providers, such as the `AppServiceProvider`:

```
User::setStripeKey('stripe-key');
```

Subscribing To A Plan

Once you have a model instance, you can easily subscribe that user to a given Stripe plan:

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

$user->subscription('monthly')->create($creditCardToken);
```

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);
```

The `subscription` 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 your plan has a trial period that is **not** configured in Stripe, you must set the trial end date manually after subscribing:

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

Specifying Additional User Details

If you would like to specify additional customer details, you may do so by passing them as 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.

No Card Up Front

If your application offers a free-trial with no credit-card up front, set the `cardUpFront` property on your model to `false`:

```
protected $cardUpFront = false;
```

On account creation, be sure to set the trial end date on the model:

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

Swapping Subscriptions

To swap a user to a new subscription, use the `swap` method:

```
$user->subscription('premium')->swap();
```

If the user is on trial, the trial will be maintained as normal. Also, if a "quantity" exists for the subscription, that quantity will also be maintained.

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);
```

Cancelling A Subscription

Cancelling a subscription is a walk in the park:

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

When a subscription is cancelled, Cashier will automatically set the `subscription_ends_at` column on 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.

Resuming A Subscription

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. Their subscription will simply be re-activated, and they will be billed on the original billing cycle.

Checking Subscription Status

To verify that a user is subscribed to your application, use the `subscribed` command:

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

The `subscribed` method makes a great candidate for a route middleware:

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed())
    {
        return redirect('billing');
    }

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

You may also determine if the user is still within their trial period (if applicable) using the `onTrial` method:

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

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 scheduled to end 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())
{
    //
}
```

The `onPlan` method may be used to determine if the user is subscribed to a given plan based on its ID:

```
if ($user->onPlan('monthly'))
{
    //
}
```

Handling Failed Payments

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 after three failed payment attempts. The `stripe/webhook` URI in this example is just for example. You will need to configure the URI in your Stripe settings.

Handling Other Stripe 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 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.

```
class WebhookController extends Laravel\Cashier\WebhookController {

    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }

}
```

> **Note:** In addition to updating the subscription information in your database, the Webhook controller will also cancel the subscription via the Stripe API.

Invoices

You can easily retrieve an array of a user's invoices using the `invoices` method:

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

When listing the invoices for the customer, you may use these helper methods to display the relevant invoice information:

```
{{ $invoice->id }}

{{ $invoice->dateString() }}

{{ $invoice->dollars() }}
```

Use the `downloadInvoice` method to generate a PDF download of the invoice. Yes, it's really this easy:

```
return $user->downloadInvoice($invoice->id, [
    'vendor'  => 'Your Company',
    'product' => 'Your Product',
]);
```

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.

Before using a Redis cache with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer.

Cache Usage

Storing An Item In The Cache

```
Cache::put('key', 'value', $minutes);
```

Using Carbon Objects To Set Expire Time

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

Storing An Item In The Cache If It Doesn't Exist

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

The `add` method will return `true` if the item is actually **added** to the cache. Otherwise, the method will return `false`.

Checking For Existence In Cache

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

Retrieving An Item From The Cache

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

Retrieving An Item Or Returning A Default Value

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

$value = Cache::get('key', function() { return 'default'; });
```

Storing An Item In The Cache Permanently

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

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. You may do this using the `Cache::remember` method:

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

You may also combine the `remember` and `forever` methods:

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

Note that all items stored in the cache are serialized, so you are free to store any type of data.

Pulling An Item From The Cache

If you need to retrieve an item from the cache and then delete it, you may use the `pull` method:

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

Removing An Item From The Cache

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

Increments & Decrements

All drivers except `file` and `database` support the `increment` and `decrement` operations:

Incrementing A Value

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

Cache::increment('key', $amount);
```

Decrementing A Value

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

Cache::decrement('key', $amount);
```

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.

Accessing A Tagged Cache

Cache tags allow you to tag related items in the cache, and then flush all caches tagged with a given name. To access a tagged cache, use the `tags` method.

You may store a tagged cache by passing in an ordered list of tag names as arguments, or as an ordered array of tag names:

```
Cache::tags('people', 'authors')->put('John', $john, $minutes);

Cache::tags(array('people', 'artists'))->put('Anne', $anne, $minutes);
```

You may use any cache storage method in combination with tags, including `remember`, `forever`, and `rememberForever`. You may also access cached items from the tagged cache, as well as use the other cache methods such as `increment` and `decrement`.

Accessing Items In A Tagged Cache

To access a tagged cache, pass the same ordered list of tags used to save it.

```
$anne = Cache::tags('people', 'artists')->get('Anne');

$john = Cache::tags(array('people', 'authors'))->get('John');
```

You may flush all items tagged with a name or list of names. 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 "John" would be removed, but not "Anne".

```
Cache::tags('authors')->flush();
```

Database Cache

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');
});
```

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:

```
$collection = collect(['taylor', 'abigail', null])->map(function($name)
{
    return strtoupper($name);
})
->reject(function($name)
{
    return is_null($value);
});
```

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, every `Collection` method returns an entirely new `Collection` instance. To dig in further, keep reading!

Basic Usage

Creating Collections

As mentioned above, the `collect` helper will return a new `Illuminate\Support\Collection` instance for the given array. You may also use the `make` command on the `Collection` class:

```
$collection = collect([1, 2, 3]);

$collection = Collection::make([1, 2, 3]);
```

Of course, collections of Eloquent objects are always returned as `Collection` instances; however, you should feel free to use the `Collection` class wherever it is convenient for your application.

Explore The Collection

Instead of listing all of the methods (there are a lot) the Collection makes available, check out the API documentation for the class!

Command Bus

Introduction

The Laravel command bus provides a convenient method of encapsulating tasks your application needs to perform into simple, easy to understand "commands". To help us understand the purpose of commands, let's pretend we are building an application that allows users to purchase podcasts.

When a user purchases a podcast, there are a variety of things that need to happen. For example, we may need to charge the user's credit card, add a record to our database that represents the purchase, and send a confirmation e-mail of the purchase. Perhaps we also need to perform some kind of validation as to whether the user is allowed to purchase podcasts.

We could put all of this logic inside a controller method; however, this has several disadvantages. The first disadvantage is that our controller probably handles several other incoming HTTP actions, and including complicated logic in each controller method will soon bloat our controller and make it harder to read. Secondly, it is difficult to re-use the purchase podcast logic outside of the controller context. Thirdly, it is more difficult to unit-test the command as we must also generate a stub HTTP request and make a full request to the application to test the purchase podcast logic.

Instead of putting this logic in the controller, we may choose to encapsulate it within a "command" object, such as a `PurchasePodcast` command.

Creating Commands

The Artisan CLI can generate new command classes using the `make:command` command:

```
php artisan make:command PurchasePodcast
```

The newly generated class will be placed in the `app/Commands` directory. By default, the command contains two methods: the constructor and the `handle` method. Of course, the constructor allows you to pass any relevant objects to the command, while the `handle` method executes the command. For example:

```
class PurchasePodcast extends Command implements SelfHandling {

    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 `handle` method may also type-hint dependencies, and they will be automatically injected by the IoC container. For example:

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

Dispatching Commands

So, once we have created a command, how do we dispatch it? Of course, we could call the `handle` method directly; however, dispatching the command through the Laravel "command bus" has several advantages which we will discuss later.

If you glance at your application's base controller, you will see the `DispatchesCommands` trait. This trait allows us to call the `dispatch` method from any of our controllers. For example:

```
public function purchasePodcast($podcastId)
{
    $this->dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
}
```

The command bus will take care of executing the command and calling the IoC container to inject any needed dependencies into the `handle` method.

You may add the `Illuminate\Foundation\Bus\DispatchesCommands` trait to any class you wish. If you would like to receive a command bus instance through the constructor of any of your classes, you may type-hint the `Illuminate\Contracts\Bus\Dispatcher` interface. Finally, you may also use the `Bus` facade to quickly dispatch commands:

```
    Bus::dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
```

Mapping Command Properties From Requests

It is very common to map HTTP request variables into commands. 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 `DispatchesCommands` trait:

```
$this->dispatchFrom('Command\Class\Name', $request);
```

This method will examine the constructor of the command class it is given, and then extract variables from the HTTP request (or any other `ArrayAccess` object) to fill the needed constructor parameters of the command. So, if our command class accepts a `firstName` variable in its constructor, the command bus will attempt to pull the `firstName` 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('Command\Class\Name', $request, [
    'firstName' => 'Taylor',
]);
```

Queued Commands

The command bus is not just for synchronous jobs that run during the current request cycle, but also serves as the primary way to build queued jobs in Laravel. So, how do we instruct command bus to queue our job for background processing instead of running it synchronously? It's easy. Firstly, when generating a new command, just add the `--queued` flag to the command:

```
php artisan make:command PurchasePodcast --queued
```

As you will see, this adds a few more features to the command, namely the `Illuminate\Contracts\Queue\ShouldBeQueued` interface and the `SerializesModels` trait. These instruct the command bus to queue the command, as well as gracefully serialize and deserialize any Eloquent models your command stores as properties.

If you would like to convert an existing command into a queued command, simply implement the `Illuminate\Contracts\Queue\ShouldBeQueued` interface on the class manually. It contains no methods, and merely serves as a "marker interface" for the dispatcher.

Then, just write your command normally. When you dispatch it to the bus that bus will automatically queue the command for background processing. It doesn't get any easier than that.

For more information on interacting with queued commands, view the full queue documentation.

Command Pipeline

Before a command is dispatched to a handler, you may pass it through other classes in a "pipeline". Command pipes work just like HTTP middleware, except for your commands! For example, a command pipe could wrap the entire command operation within a database transaction, or simply log its execution.

To add a pipe to your bus, call the `pipeThrough` method of the dispatcher from your `App\Providers\BusServiceProvider::boot` method:

```
$dispatcher->pipeThrough(['UseDatabaseTransactions', 'LogCommand']);
```

A command pipe is defined with a `handle` method, just like a middleware:

```
class UseDatabaseTransactions {

    public function handle($command, $next)
    {
        return DB::transaction(function() use ($command, $next)
        {
            return $next($command);
```

```
        }
    }

}
```

Command pipe classes are resolved through the IoC container, so feel free to type-hint any dependencies you need within their constructors.

You may even define a `Closure` as a command pipe:

```
$dispatcher->pipeThrough([function($command, $next)
{
    return DB::transaction(function() use ($command, $next)
    {
        return $next($command);
    }
}]);
```

Extending The Framework

Managers & Factories

Laravel has several `Manager` classes that manage the creation of driver-based components. These include the cache, session, authentication, and queue components. The manager class is responsible for creating a particular driver implementation based on the application's configuration. For example, the `CacheManager` class can create APC, Memcached, File, and various other implementations of cache drivers.

Each of these managers includes an `extend` method which may be used to easily inject new driver resolution functionality into the manager. We'll cover each of these managers below, with examples of how to inject custom driver support into each of them.

> **Note:** Take a moment to explore the various `Manager` classes that ship with Laravel, such as the `CacheManager` and `SessionManager`. Reading through these classes will give you a more thorough understanding of how Laravel works under the hood. All manager classes extend the `Illuminate\Support\Manager` base class, which provides some helpful, common functionality for each manager.

Cache

To extend the Laravel cache facility, we will use the `extend` method on the `CacheManager`, which is used to bind a custom driver resolver to the manager, and is common across all manager classes. For example, to register a new cache driver named "mongo", we would do the following:

```
Cache::extend('mongo', function($app)
{
    return Cache::repository(new MongoStore);
});
```

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 `Illuminate\Foundation\Application` and an IoC 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:

```
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() {}

}
```

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);
});
```

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.

Session

Extending Laravel with a custom session driver is just as easy as extending the cache system. Again, we will use the `extend` method to register our custom code:

```
Session::extend('mongo', function($app)
{
    // Return implementation of SessionHandlerInterface
});
```

Where To Extend The Session

You should place your session extension code in the `boot` method of your `AppServiceProvider`.

Writing The Session Extension

Note that our custom session driver should implement the `SessionHandlerInterface`. This interface contains just a few simple methods we need to implement. A stubbed MongoDB implementation would look something like this:

```
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 `SessionHandlerInterface` has been implemented, we are ready to register it with the Session manager:

```
Session::extend('mongo', function($app)
{
    return new MongoHandler;
});
```

Once the session driver has been registered, we may use the `mongo` driver in our `config/session.php` configuration file.

> **Note:** Remember, if you write a custom session handler, share it on Packagist!

Authentication

Authentication may be extended the same way as the cache and session facilities. Again, we will use the `extend` method we have become familiar with:

```
Auth::extend('riak', function($app)
{
    // Return implementation of Illuminate\Contracts\Auth\UserProvider
});
```

The `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 `UserProvider` contract:

```
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 numeric 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 with 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 successfull "remember me" login attempt, or a null when 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']`. **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']`.

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable`. Remember, the provider should return implementations of this interface from the `retrieveById` and `retrieveByCredentials` methods:

```
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.

Finally, once we have implemented the `UserProvider`, we are ready to register our extension with the `Auth` facade:

```
Auth::extend('riak', function($app)
{
    return new RiakUserProvider($app['riak.connection']);
});
```

After you have registered the driver with the `extend` method, you switch to the new driver in your `config/auth.php` configuration file.

IoC Based Extension

Almost every service provider included with the Laravel framework binds objects into the IoC container. You can find a list of your application's service providers in the `config/app.php` configuration file. As you have time, you should skim through each of these provider's source code. By doing so, you will gain a much better understanding of what each provider adds to the framework, as well as what keys are used to bind various services into the IoC container.

For example, the `HashServiceProvider` binds a `hash` key into the IoC container, which resolves into a `Illuminate\Hashing\BcryptHasher` instance. You can easily extend and override this class within your own application by overriding this IoC binding. For example:

```
<?php namespace App\Providers;

class SnappyHashProvider extends \Illuminate\Hashing\HashServiceProvider {

    public function boot()
    {
        $this->app->bindShared('hash', function()
        {
            return new \Snappy\Hashing\ScryptHasher;
        });

        parent::boot();
    }

}
```

Note that this class extends the `HashServiceProvider`, not the default `ServiceProvider` base class. Once you have

extended the service provider, swap out the `HashServiceProvider` in your `config/app.php` configuration file with the name of your extended provider.

This is the general method of extending any core class that is bound in the container. Essentially every core class is bound in the container in this fashion, and can be overridden. Again, reading through the included framework service providers will familiarize you with where various classes are bound into the container, and what keys they are bound by. This is a great way to learn more about how Laravel is put together.

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.

If you've ever been confused about how to get started with Gulp and asset compilation, you will love Laravel Elixir!

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. Don't worry, it's quick and easy!

Gulp

Next, you'll want to pull in Gulp as a global NPM package like so:

```
npm install --global gulp
```

Laravel Elixir

The only remaining step is to install Elixir! With a new install 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
```

Usage

Now that you've installed Elixir, you'll be compiling and concatenating in no time!

Compile Less

```
elixir(function(mix) {
    mix.less("app.less");
```

```
});
```

In the example above, Elixir assumes that your Less files are stored in `resources/assets/less`.

Compile Sass

```
elixir(function(mix) {
    mix.sass("app.sass");
});
```

This assumes that your Sass files are stored in `resources/assets/sass`.

Compile CoffeeScript

```
elixir(function(mix) {
    mix.coffee();
});
```

This assumes that your CoffeeScript files are stored in `resources/assets/coffee`.

Compile All Less and CoffeeScript

```
elixir(function(mix) {
    mix.less()
       .coffee();
});
```

Trigger PHPUnit Tests

```
elixir(function(mix) {
    mix.phpUnit();
});
```

Trigger PHPSpec Tests

```
elixir(function(mix) {
    mix.phpSpec();
});
```

Combine Stylesheets

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ]);
});
```

Paths passed to this method are relative to the `resources/css` directory.

Combine Stylesheets and Save to a Custom Directory

```
elixir(function(mix) {
```

```
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css');
});
```

Combine Stylesheets From A Custom Base Directory

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css', 'public/css');
});
```

The third argument to both the `styles` and `scripts` methods determines the relative directory for all paths passed to the methods.

Combine All Styles in a Directory

```
elixir(function(mix) {
    mix.stylesIn("public/css");
});
```

Combine Scripts

```
elixir(function(mix) {
    mix.scripts([
        "jquery.js",
        "app.js"
    ]);
});
```

Again, this assumes all paths are relative to the `resources/js` directory.

Combine All Scripts in a Directory

```
elixir(function(mix) {
    mix.scriptsIn("public/js/some/directory");
});
```

Combine Multiple Sets of Scripts

```
elixir(function(mix) {
    mix.scripts(['jquery.js', 'main.js'], 'public/js/main.js')
       .scripts(['forum.js', 'threads.js'], 'public/js/forum.js');
});
```

Version / Hash A File

```
elixir(function(mix) {
    mix.version("css/all.css");
});
```

This 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`.

Within your views, you may use the `elixir()` function to load the appropriately hashed asset. Here's an example:

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
```

Behind the scenes, the `elixir()` function will determine the name of the hashed file that should be included. Don't you feel the weight lifting off your shoulders already?

Copy a File to a New Location

```
elixir(function(mix) {
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');
});
```

Copy an Entire Directory to a New Location

```
elixir(function(mix) {
    mix.copy('vendor/package/views', 'resources/views');
});
```

Method Chaining

Of course, you may chain almost all of Elixir's methods together to build your recipe:

```
elixir(function(mix) {
    mix.less("app.less")
       .coffee()
       .phpUnit()
       .version("css/bootstrap.css");
});
```

Gulp

Now that you've told Elixir which tasks to execute, you only need to trigger Gulp from the command line.

Execute All Registered Tasks Once

```
gulp
```

Watch Assets For Changes

```
gulp watch
```

Watch Tests And PHP Classes for Changes

```
gulp tdd
```

> **Note:** All tasks will assume a development environment, and will exclude minification. For production, use `gulp --`

> `production`.

Extensions

You can even create your own Gulp tasks, and hook them into Elixir. Imagine that you want to add a fun task that uses the Terminal to verbally notify you with some message. Here's what that might look like:

```
var gulp = require("gulp");
var shell = require("gulp-shell");
var elixir = require("laravel-elixir");

elixir.extend("message", function(message) {

    gulp.task("say", function() {
        gulp.src("").pipe(shell("say " + message));
    });

    return this.queueTask("say");

});
```

Notice that we `extend` Elixir's API by passing the key that we will use within our Gulpfile, as well as a callback function that will create the Gulp task.

If you want your custom task to be monitored, then register a watcher as well.

```
this.registerWatcher("message", "**/*.php");
```

This lines designates that when any file that matches the regex, `**/*.php` is modified, we want to trigger the `message` task.

That's it! You may either place this at the top of your Gulpfile, or instead extract it to a custom tasks file. If you choose the latter approach, simply require it into your Gulpfile, like so:

```
require("./custom-tasks")
```

You're done! Now, you can mix it in.

```
elixir(function(mix) {
    mix.message("Tea, Earl Grey, Hot");
});
```

With this addition, each time you trigger Gulp, Picard will request some tea.

Encryption

Introduction

Laravel provides facilities for strong AES encryption via the Mcrypt PHP extension.

Basic Usage

Encrypting A Value

```
$encrypted = Crypt::encrypt('secret');
```

> **Note:** Be sure to set a 16, 24, or 32 character random string in the `key` option of the `config/app.php` file. Otherwise, encrypted values will not be secure.

Decrypting A Value

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Setting The Cipher & Mode

You may also set the cipher and mode used by the encrypter:

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

Errors & Logging

Configuration

The logging facilities for your application are configured in the `Illuminate\Foundation\Bootstrap\ConfigureLogging` bootstrapper class. This class utilizes the `log` configuration option from your `config/app.php` configuration file.

By default, the logger is configured to use daily log files; however, you may customize this behavior as needed. Since Laravel uses the popular Monolog logging library, you can take advantage of the variety of handlers that Monolog offers.

For example, if you wish to use a single log file instead of daily files, you can make the following change to your `config/app.php` configuration file:

```
'log' => 'single'
```

Out of the box, Laravel supported `single`, `daily`, and `syslog` logging modes. However, you are free to customize the logging for your application as you wish by overriding the `ConfigureLogging` bootstrapper class.

Error Detail

The amount of error detail your application displays through the browser is controlled by the `app.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`.**

Handling Errors

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`.

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 implementation on the parent class where the exception is logged. However, you are free to log exceptions however you wish. If you need to report different types of exceptions in different ways, you may use the 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);
    }
```

The `render` method is responsible for converting the 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.

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.

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 return such a response, use the following:

```
abort(404);
```

Optionally, you may provide a response:

```
abort(403, 'Unauthorized action.');
```

This method may be used at any time during the request's lifecycle.

Custom 404 Error Page

To return a custom view for all 404 errors, create a `resources/views/errors/404.blade.php` file. This view will be served on all 404 errors generated by your application.

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 log like so:

```
Log::info('This is some useful information.');

Log::warning('Something could be going wrong.');

Log::error('Something is really going wrong.');
```

The logger provides the seven logging levels defined in RFC 5424: **debug**, **info**, **notice**, **warning**, **error**, **critical**, and **alert**.

An array of contextual data may also be passed to the log methods:

```
Log::info('Log message', ['context' => 'Other helpful information']);
```

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();
```

You may also register an event to catch all messages passed to the log:

Registering A Log Event Listener

```
Log::listen(function($level, $message, $context)
{
    //
});
```

Events

Basic Usage

The Laravel event facilities 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 handlers are stored in `app/Handlers/Events`.

You can generate a new event class using the Artisan CLI tool:

```
php artisan make:event PodcastWasPurchased
```

Subscribing To An Event

The `EventServiceProvider` included with your Laravel application provides a convenient place to register all event handlers. The `listen` property contains an array of all events (keys) and their handlers (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 handler mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Handlers\Events\EmailPurchaseConfirmation@handle',
    ],
];
```

To generate a handler for an event, use the `handler:event` Artisan CLI command:

```
php artisan handler:event EmailPurchaseConfirmation --event=PodcastWasPurchased
```

Of course, manually running the `make:event` and `handler:event` commands each time you need a handler or event is cumbersome. Instead, simply add handlers and events to your `EventServiceProvider` and use the `event:generate` command. This command will generate any events or handlers that are listed in your `EventServiceProvider`:

```
php artisan event:generate
```

Firing An Event

Now we are ready to fire our event using the `Event` facade:

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

The `fire` method returns an array of responses that you can use to control what happens next in your application.

You may also use the `event` helper to fire an event:

```
event(new PodcastWasPurchased($podcast));
```

Closure Listeners

You can even listen to events without creating a separate handler class at all. For example, in the `boot` method of your `EventServiceProvider`, you could do the following:

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // Handle the event...
});
```

Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so using by returning `false` from your handler:

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // Handle the event...

    return false;
});
```

Queued Event Handlers

Need to queue an event handler? It couldn't be any easier. When generating the handler, simply use the `--queued` flag:

```
php artisan handler:event SendPurchaseConfirmation --event=PodcastWasPurchased --queued
```

This will generate a handler class that implements the `Illuminate\Contracts\Queue\ShouldBeQueued` interface. That's it! Now when this handler is called for an event, it will be queued automatically by the event dispatcher.

If no exceptions are thrown when the handler is executed by the queue, the queued job will be deleted automatically after it has processed. If you need to access the queued job's `delete` and `release` methods manually, you may do so. The `Illuminate\Queue\InteractsWithQueue` trait, which is included by default on queued handlers, gives you access to these methods:

```
public function handle(PodcastWasPurchased $event)
{
    if (true)
    {
        $this->release(30);
    }
}
```

If you have an existing handler that you would like to convert to a queued handler, simply add the `ShouldBeQueued` interface to the class manually.

Event Subscribers

Defining An Event Subscriber

Event subscribers are classes that may subscribe to multiple events from within the class itself. Subscribers should define a `subscribe` method, which will be passed an event dispatcher instance:

```
class UserEventHandler {

    /**
     * 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  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen('App\Events\UserLoggedIn', 'UserEventHandler@onUserLogin');

        $events->listen('App\Events\UserLoggedOut', 'UserEventHandler@onUserLogout');
    }

}
```

Registering An Event Subscriber

Once the subscriber has been defined, it may be registered with the `Event` class.

```
$subscriber = new UserEventHandler;

Event::subscribe($subscriber);
```

You may also use the Laravel IoC container to resolve your subscriber. To do so, simply pass the name of your subscriber to the `subscribe` method:

```
Event::subscribe('UserEventHandler');
```

Filesystem / Cloud Storage

Introduction

Laravel provides a wonderful filesystem abstraction thanks to the 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!

Before using the S3 or Rackspace drivers, you will need to install the appropriate package via Composer:

- Amazon S3: `league/flysystem-aws-s3-v2 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

Of course, you may configure as many disks as you like, and may even have multiple disks that use the same 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');
```

Basic Usage

The `Storage` facade may be used to interact with any of your configured disks. Alternatively, you may type-hint the `Illuminate\Contracts\Filesystem\Factory` contract on any class that is resolved via the IoC container.

Retrieving A Particular Disk

```
$disk = Storage::disk('s3');

$disk = Storage::disk('local');
```

Determining If A File Exists

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

Calling Methods On The Default Disk

```
if (Storage::exists('file.jpg'))
{
    //
}
```

Retrieving A File's Contents

```
$contents = Storage::get('file.jpg');
```

Setting A File's Contents

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

Prepend To A File

```
Storage::prepend('file.log', 'Prepended Text');
```

Append To A File

```
Storage::append('file.log', 'Appended Text');
```

Delete A File

```
Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

Copy A File To A New Location

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

Move A File To A New Location

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Get File Size

```
$size = Storage::size('file1.jpg');
```

Get The Last Modification Time (UNIX)

```
$time = Storage::lastModified('file1.jpg');
```

Get All Files Within A Directory

```
$files = Storage::files($directory);

// Recursive...
$files = Storage::allFiles($directory);
```

Get All Directories Within A Directory

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

Create A Directory

```
Storage::makeDirectory($directory);
```

Delete A Directory

```
Storage::deleteDirectory($directory);
```

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 be take care of verifying the Bcrypt password against the un-hashed version provided by the user.

Likewise, the user `Registrar` service that ships with Laravel makes the proper `bcrypt` function call to hash stored passwords.

Basic Usage

Hashing A Password Using Bcrypt

```
$password = Hash::make('secret');
```

You may also use the `bcrypt` helper function:

```
$password = bcrypt('secret');
```

Verifying A Password Against A Hash

```
if (Hash::check('secret', $hashedPassword))
{
    // The passwords match...
}
```

Checking If A Password Needs To Be Rehashed

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

Helper Functions

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('foo' => 'bar');

$array = array_add($array, 'key', 'value');
```

array_divide

The `array_divide` function returns two arrays, one containing the keys, and the other containing the values of the original array.

```
$array = array('foo' => 'bar');

list($keys, $values) = array_divide($array);
```

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('foo' => array('bar' => 'baz'));

$array = array_dot($array);

// array('foo.bar' => 'baz');
```

array_except

The `array_except` method removes the given key / value pairs from the array.

```
$array = array_except($array, array('keys', 'to', 'remove'));
```

array_fetch

The `array_fetch` method returns a flattened array containing the selected nested element.

```
$array = array(
    array('developer' => array('name' => 'Taylor')),
    array('developer' => array('name' => 'Dayle')),
);
```

```
$array = array_fetch($array, 'developer.name');

// array('Taylor', 'Dayle');
```

array_first

The `array_first` method returns the first element of an array passing a given truth test.

```
$array = array(100, 200, 300);

$value = array_first($array, function($key, $value)
{
    return $value >= 150;
});
```

A default value may also be passed as the third parameter:

```
$value = array_first($array, $callback, $default);
```

array_last

The `array_last` method returns the last element of an array passing a given truth test.

```
$array = array(350, 400, 500, 300, 200, 100);

$value = array_last($array, function($key, $value)
{
    return $value > 350;
});

// 500
```

A default value may also be passed as the third parameter:

```
$value = array_last($array, $callback, $default);
```

array_flatten

The `array_flatten` method will flatten a multi-dimensional array into a single level.

```
$array = array('name' => 'Joe', 'languages' => array('PHP', 'Ruby'));

$array = array_flatten($array);

// array('Joe', 'PHP', 'Ruby');
```

array_forget

The `array_forget` method will remove a given key / value pair from a deeply nested array using "dot" notation.

```
$array = array('names' => array('joe' => array('programmer')));

array_forget($array, 'names.joe');
```

array_get

The `array_get` method will retrieve a given value from a deeply nested array using "dot" notation.

```
$array = array('names' => array('joe' => array('programmer')));

$value = array_get($array, 'names.joe');

$value = array_get($array, 'names.john', 'default');
```

> **Note:** Want something like `array_get` but for objects instead? Use `object_get`.

array_only

The `array_only` method will return only the specified key / value pairs from the array.

```
$array = array('name' => 'Joe', 'age' => 27, 'votes' => 1);

$array = array_only($array, array('name', 'votes'));
```

array_pluck

The `array_pluck` method will pluck a list of the given key / value pairs from the array.

```
$array = array(array('name' => 'Taylor'), array('name' => 'Dayle'));

$array = array_pluck($array, 'name');

// array('Taylor', 'Dayle');
```

array_pull

The `array_pull` method will return a given key / value pair from the array, as well as remove it.

```
$array = array('name' => 'Taylor', 'age' => 27);

$name = array_pull($array, 'name');
```

array_set

The `array_set` method will set a value within a deeply nested array using "dot" notation.

```
$array = array('names' => array('programmer' => 'Joe'));

array_set($array, 'names.editor', 'Taylor');
```

array_sort

The `array_sort` method sorts the array by the results of the given Closure.

```
$array = array(
    array('name' => 'Jill'),
    array('name' => 'Barry'),
);

$array = array_values(array_sort($array, function($value)
{
    return $value['name'];
}));
```

array_where

Filter the array using the given Closure.

```
$array = array(100, '200', 300, '400', 500);

$array = array_where($array, function($key, $value)
{
    return is_string($value);
});

// Array ( [1] => 200 [3] => 400 )
```

head

Return the first element in the array. Useful for method chaining in PHP 5.3.x.

```
$first = head($this->returnsArray('foo'));
```

last

Return the last element in the array. Useful for method chaining.

```
$last = last($this->returnsArray('foo'));
```

Paths

app_path

Get the fully qualified path to the `app` directory.

```
$path = app_path();
```

base_path

Get the fully qualified path to the root of the application install.

public_path

Get the fully qualified path to the `public` directory.

storage_path

Get the fully qualified path to the `storage` directory.

Strings

camel_case

Convert the given string to `camelCase`.

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename

Get the class name of the given class, without any namespace names.

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e

Run `htmlentities` over the given string, with UTF-8 support.

```
$entities = e('<html>foo</html>');
```

ends_with

Determine if the given haystack ends with a given needle.

```
$value = ends_with('This is my name', 'name');
```

snake_case

Convert the given string to `snake_case`.

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit

Limit the number of characters in a string.

```
str_limit($value, $limit = 100, $end = '...')
```

Example:

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with

Determine if the given haystack begins with the given needle.

```
$value = starts_with('This is my name', 'This');
```

str_contains

Determine if the given haystack contains the given needle.

```
$value = str_contains('This is my name', 'my');
```

str_finish

Add a single instance of the given needle to the haystack. Remove any extra instances.

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is

Determine if a given string matches a given pattern. Asterisks may be used to indicate wildcards.

```
$value = str_is('foo*', 'foobar');
```

str_plural

Convert a string to its plural form (English only).

```
$plural = str_plural('car');
```

str_random

Generate a random string of the given length.

```
$string = str_random(40);
```

str_singular

Convert a string to its singular form (English only).

```
$singular = str_singular('cars');
```

str_slug

Generate a URL friendly "slug" from a given string.

```
str_slug($title, $separator);
```

Example:

```
$title = str_slug("Laravel 5 Framework", "-");
```

```
// laravel-5-framework
```

studly_case

Convert the given string to `StudlyCase`.

```
$value = studly_case('foo_bar');

// FooBar
```

trans

Translate a given language line. Alias of `Lang::get`.

```
$value = trans('validation.required'):
```

trans_choice

Translate a given language line with inflection. Alias of `Lang::choice`.

```
$value = trans_choice('foo.bar', $count);
```

URLs

action

Generate a URL for a given controller action.

```
$url = action('HomeController@getIndex', $params);
```

route

Generate a URL for a given named route.

```
$url = route('routeName', $params);
```

asset

Generate a URL for an asset.

```
$url = asset('img/photo.jpg');
```

link_to

Generate a HTML link to the given URL.

```
echo link_to('foo/bar', $title, $attributes = array(), $secure = null);
```

link_to_asset

Generate a HTML link to the given asset.

```
echo link_to_asset('foo/bar.zip', $title, $attributes = array(), $secure = null);
```

link_to_route

Generate a HTML link to the given route.

```
echo link_to_route('route.name', $title, $parameters = array(), $attributes = array());
```

link_to_action

Generate a HTML link to the given controller action.

```
echo link_to_action('HomeController@getIndex', $title, $parameters = array(), $attributes = array());
```

secure_asset

Generate a HTML link to the given asset using HTTPS.

```
echo secure_asset('foo/bar.zip', $title, $attributes = array());
```

secure_url

Generate a fully qualified URL to a given path using HTTPS.

```
echo secure_url('foo/bar', $parameters = array());
```

url

Generate a fully qualified URL to the given path.

```
echo url('foo/bar', $parameters = array(), $secure = null);
```

Miscellaneous

csrf_token

Get the value of the current CSRF token.

```
$token = csrf_token();
```

dd

Dump the given variable and end execution of the script.

```
dd($value);
```

value

If the given value is a `Closure`, return the value returned by the `Closure`. Otherwise, return the value.

```
$value = value(function() { return 'bar'; });
```

with

Return the given object. Useful for method chaining constructors in PHP 5.3.x.

```
$value = with(new Foo)->doWork();
```

Localization

Introduction

The Laravel `Lang` facade provides a convenient way of retrieving strings in various languages, allowing you to easily support multiple languages within your application.

Language Files

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
```

Example Language File

Language files simply return an array of keyed strings. For example:

```
<?php

return array(
    'welcome' => 'Welcome to our application'
);
```

Changing The Default Language At Runtime

The default language for your application is stored in the `config/app.php` configuration file. You may change the active language at any time using the `App::setLocale` method:

```
App::setLocale('es');
```

Setting The Fallback Language

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

Retrieving Lines From A Language File

```
echo Lang::get('messages.welcome');
```

The first segment of the string passed to the `get` method is the name of the language file, and the second is the name of the line that should be retrieved.

> **Note:** If a language line does not exist, the key will be returned by the `get` method.

You may also use the `trans` helper function, which is an alias for the `Lang::get` method.

```
echo trans('messages.welcome');
```

Making Replacements In Lines

You may also define place-holders in your language lines:

```
'welcome' => 'Welcome, :name',
```

Then, pass a second argument of replacements to the `Lang::get` method:

```
echo Lang::get('messages.welcome', array('name' => 'Dayle'));
```

Determine If A Language File Contains A Line

```
if (Lang::has('messages.welcome'))
{
    //
}
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization. You may easily manage this in your language files. By using a "pipe" character, you may separate the singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

You may then use the `Lang::choice` method to retrieve the line:

```
echo Lang::choice('messages.apples', 10);
```

You may also supply a locale argument to specify the language. For example, if you want to use the Russian (ru) language:

```
echo Lang::choice('товар|товара|товаров', $count, array(), 'ru');
```

Since the Laravel translator is powered by the Symfony Translation component, you may also create more explicit pluralization rules easily:

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

Validation

For localization for validation errors and messages, take a look at the documentation on Validation.

Overriding Package Language Files

Many packages ship with their own language lines. Instead of hacking the package's core files to tweak these lines, you may override them by placing files in the `resources/lang/packages/{locale}/{package}` 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/packages/en/hearthfire/messages.php`. In this file you would define only the language lines you wish to override. Any language lines you don't override will still be loaded from the package's language files.

Mail

Configuration

Laravel provides a clean, simple API over the popular SwiftMailer library. The mail configuration file is `config/mail.php`, and contains options allowing you to change your SMTP host, port, and credentials, as well as set a global `from` address for all messages delivered by the library. You may use any SMTP server you wish. If you wish to use the PHP `mail` function to send mail, you may change the `driver` to `mail` in the configuration file. A `sendmail` driver is also available.

API Drivers

Laravel also includes drivers for the Mailgun and Mandrill HTTP APIs. These APIs are often simpler and quicker than the SMTP servers. Both of these drivers require that the Guzzle 4 HTTP library be installed into your application. You can add Guzzle 4 to your project by adding the following line to your `composer.json` file:

```
"guzzlehttp/guzzle": "~4.0"
```

Mailgun Driver

To use the Mailgun driver, set the `driver` option to `mailgun` in your `config/mail.php` configuration file. Next, create an `config/services.php` configuration file if one does not already exist for your project. Verify that it contains the following options:

```
'mailgun' => array(
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
),
```

Mandrill Driver

To use the Mandrill driver, set the `driver` option to `mandrill` in your `config/mail.php` configuration file. Next, create an `config/services.php` configuration file if one does not already exist for your project. Verify that it contains the following options:

```
'mandrill' => array(
    'secret' => 'your-mandrill-key',
),
```

Log Driver

If the `driver` option of your `config/mail.php` configuration file is set to `log`, all e-mails will be written to your log files, and will not actually be sent to any of the recipients. This is primarily useful for quick, local debugging and content verification.

Basic Usage

The `Mail::send` method may be used to send an e-mail message:

```
Mail::send('emails.welcome', array('key' => 'value'), function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

The first argument passed to the `send` method is the name of the view that should be used as the e-mail body. The second is the data to be passed to the view, often as an associative array where the data items are available to the view by `$key`. The third is a Closure allowing you to specify various options on the e-mail message.

> **Note:** A `$message` variable is always passed to e-mail views, and allows the inline embedding of attachments. So, it is best to avoid passing a `message` variable in your view payload.

You may also specify a plain text view to use in addition to an HTML view:

```
Mail::send(array('html.view', 'text.view'), $data, $callback);
```

Or, you may specify only one type of view using the `html` or `text` keys:

```
Mail::send(array('text' => 'view'), $data, $callback);
```

You may specify other options on the e-mail message such as any carbon copies or attachments as well:

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');

    $message->attach($pathToFile);
});
```

When attaching files to a message, you may also specify a MIME type and / or a display name:

```
$message->attach($pathToFile, array('as' => $display, 'mime' => $mime));
```

If you just need to e-mail a simple string instead of an entire view, use the `raw` method:

```
Mail::raw('Text to e-mail', function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');
});
```

> **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.

Embedding Inline Attachments

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.

Embedding An Image In An E-Mail View

```
<body>
    Here is an image:

    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

Embedding Raw Data In An E-Mail View

```
<body>
    Here is an image from raw data:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

Note that the `$message` variable is always passed to e-mail views by the `Mail` facade.

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, simply use the `queue` method on the `Mail` facade:

```
Mail::queue('emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

You may also specify the number of seconds you wish to delay the sending of the mail message using the `later` method:

```
Mail::later(5, 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

If you wish to specify a specific queue or "tube" 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)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

Mail & Local Development

When developing an application that sends e-mail, it's usually desirable to disable the sending of messages from your local or development environment. To do so, you may either call the `Mail::pretend` method, or set the `pretend` option in the `config/mail.php` configuration file to `true`. When the mailer is in `pretend` mode, messages will be written to your application's log files instead of being sent to the recipient.

If you would like to actually view the test e-mails, consider using a service like MailTrap.

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 that are Laravel specific.

All Laravel packages are distributed via Packagist and Composer, so learning about these wonderful PHP package distribution tools is essential.

Views

Your package's internal structure is entirely up to you; however, typically each package will contain one or more service providers. The service provider contains any IoC bindings, as well as instructions as to where package configuration, views, and translation files are located.

Views

Package views are typically referenced using a double-colon "namespace" syntax:

```
return view('package::view.name');
```

All you need to do is tell Laravel where the views for a given namespace are located. For example, if your package is named "courier", you might add the following to your service provider's `boot` method:

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

Now you may load your package views using the following syntax:

```
return view('courier::view.name');
```

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

To publish your package's views to the `resources/views/vendor` directory, you should use the `publishes` method from the `boot` method of your service provider:

```
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` command, your views directory will be copied to the specified location.

If you would like to overwrite existing files, use the `--force` switch:

```
php artisan vendor:publish --force
```

> **Note:** You may use the `publishes` method to publish **any** type of file to any location you wish.

Translations

Package translation files are typically referenced using a double-colon syntax:

```
return trans('package::file.line');
```

All you need to do is tell Laravel where the translations for a given namespace are located. For example, if your package is named "courier", you might add the following to your service provider's `boot` method:

```
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

Note that within your `translations` folder, you would have further directories for each language, such as `en`, `es`, `ru`, etc.

Now you may load your package translations using the following syntax:

```
return trans('courier::file.line');
```

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:

```
$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');
```

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:

```
$this->mergeConfigFrom(
    __DIR__.'/path/to/config/courier.php', 'courier'
);
```

Publishing File Groups

You may want to publish groups of files separately. For instance, you might want your users to be able to publish your package's configuration files and asset files separately. You can do this by 'tagging' them:

```
// Publish a config file
$this->publishes([
    __DIR__.'/../config/package.php', config_path('package.php')
], 'config');

// Publish your migrations
$this->publishes([
    __DIR__.'/../database/migrations/' => base_path('/database/migrations')
], 'migrations');
```

You can then publish these files separately by referencing their tag like so:

```
php artisan vendor:publish --provider="Vendor\Providers\PackageServiceProvider" --tag="config"
```

Routing

To load a routes file for your package, simply `include` it from within your service provider's `boot` method.

Including A Routes File From A Service Provider

```
public function boot()
{
    include __DIR__.'/../../routes.php';
}
```

> **Note:** If your package is using controllers, you will need to make sure they are properly configured in your `composer.json` file's auto-load section.

Pagination

Configuration

In other frameworks, pagination can be very painful. Laravel makes it a breeze. Laravel can generate an intelligent "range" of links based on the current page. The generated HTML is compatible with the Bootstrap CSS framework.

Usage

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent model.

Paginating Database Results

```
$users = DB::table('users')->paginate(15);
```

> **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.

Creating A Paginator Manually

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.

Paginating An Eloquent Model

You may also paginate Eloquent models:

```
$allUsers = User::paginate(15);

$someUsers = User::where('votes', '>', 100)->paginate(15);
```

The argument passed to the `paginate` method is the number of items you wish to display per page. Once you have retrieved the results, you may display them on your view, and create the pagination links using the `render` method:

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->render(); ?>
```

This is all it takes to create a pagination system! Note that we did not have to inform the framework of the current page. Laravel will determine this for you automatically.

You may also access additional pagination information via the following methods:

- `currentPage`
- `lastPage`
- `perPage`
- `total`
- `count`

"Simple Pagination"

If you are only showing "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 useful for larger datasets when you do not require the display of exact page numbers on your view:

```
$someUsers = User::where('votes', '>', 100)->simplePaginate(15);
```

Customizing The Paginator URI

You may also customize the URI used by the paginator via the `setPath` method:

```
$users = User::paginate();

$users->setPath('custom/url');
```

The example above will create URLs like the following: http://example.com/custom/url?page=2

Appending To Pagination Links

You can add to the query string of pagination links using the `appends` method on the Paginator:

```
<?php echo $users->appends(['sort' => 'votes'])->render(); ?>
```

This will generate URLs that look something like this:

```
http://example.com/something?page=2&sort=votes
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the `fragment` method:

```
<?php echo $users->fragment('foo')->render(); ?>
```

This method call will generate URLs that look something like this:

```
http://example.com/something?page=2#foo
```

Converting To JSON

The `Paginator` class implements the `Illuminate\Contracts\Support\JsonableInterface` contract and exposes the `toJson` method. You may also convert a `Paginator` instance to JSON by returning it from a route. The JSON'd form of the instance will include some "meta" information such as `total`, `current_page`, and `last_page`. The instance's data will be available

via the `data` key in the JSON array.

Queues

Configuration

The Laravel Queue component provides a unified API across a variety of different queue services. Queues allow you to defer the processing of a time consuming task, such as sending an e-mail, until a later time, thus drastically speeding up the web requests to your application.

The queue configuration file is stored in `config/queue.php`. In this file you will find connection configurations for each of the queue drivers that are included with the framework, which includes a database, Beanstalkd, IronMQ, Amazon SQS, Redis, null, and synchronous (for local use) driver. The `null` queue driver simply discards queued jobs so they are never run.

Queue Database Table

In order to use the `database` queue driver, you will need a database table to hold the jobs. To generate a migration to create this table, run the `queue:table` Artisan command:

```
php artisan queue:table
```

Other Queue Dependencies

The following dependencies are needed for the listed queue drivers:

- Amazon SQS: `aws/aws-sdk-php`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq`
- Redis: `predis/predis ~1.0`

Basic Usage

Pushing A Job Onto The Queue

All of the queueable jobs for your application are stored in the `App\Commands` directory. You may generate a new queued command using the Artisan CLI:

```
php artisan make:command SendEmail --queued
```

To push a new job onto the queue, use the `Queue::push` method:

```
Queue::push(new SendEmail($message));
```

> **Note:** In this example, we are using the `Queue` facade directly; however, typically you would dispatch queued command via the Command Bus. We will continue to use the `Queue` facade throughout this page; however, familiarize with the command bus as well, since it is used to dispatch both queued and synchronous commands for your application.

By default, the `make:command` Artisan command generates a "self-handling" command, meaning a `handle` method is added to the command itself. This method will be called when the job is executed by the queue. You may type-hint any dependencies you need on the `handle` method and the IoC container will automatically inject them:

```
public function handle(UserRepository $users)
{
    //
}
```

If you would like your command to have a separate handler class, you should add the `--handler` flag to the `make:command` command:

```
php artisan make:command SendEmail --queued --handler
```

The generated handler will be placed in `App\Handlers\Commands` and will be resolved out of the IoC container.

Specifying The Queue / Tube For A Job

You may also specify the queue / tube a job should be sent to:

```
Queue::pushOn('emails', new SendEmail($message));
```

Passing The Same Payload To Multiple Jobs

If you need to pass the same data to several queue jobs, you may use the `Queue::bulk` method:

```
Queue::bulk(array(new SendEmail($message), new AnotherCommand));
```

Delaying The Execution Of A Job

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 an e-mail 15 minutes after sign-up. You can accomplish this using the `Queue::later` method:

```
$date = Carbon::now()->addMinutes(15);

Queue::later($date, new SendEmail($message));
```

In this example, we're using the Carbon date library to specify the delay we wish to assign to the job. Alternatively, you may pass the number of seconds you wish to delay as an integer.

> **Note:** The Amazon SQS service has a delay limit of 900 seconds (15 minutes).

Queues And Eloquent Models

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.

Deleting A Processed Job

Once you have processed a job, it must be deleted from the queue. If no exception is thrown during the execution of your job, this will be done automatically.

If you would like to `delete` or `release` the job manually, the `Illuminate\Queue\InteractsWithQueue` trait provides access to the queue job `release` and `delete` methods. The `release` method accepts a single value: the number of seconds you wish to wait until the job is made available again.

```
public function handle(SendEmail $command)
{
    if (true)
    {
        $this->release(30);
    }
}
```

Releasing A Job Back Onto The Queue

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 commands.

Checking The Number Of Run Attempts

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:

```
if ($this->attempts() > 3)
{
    //
}
```

> **Note:** Your command / handler must use the `Illuminate\Queue\InteractsWithQueue` trait in order to call this method.

Queueing Closures

You may also push a Closure onto the queue. This is very convenient for quick, simple tasks that need to be queued:

Pushing A Closure Onto The Queue

```
Queue::push(function($job) use ($id)
{
    Account::delete($id);

    $job->delete();
});
```

> **Note:** Instead of making objects available to queued Closures via the `use` directive, consider passing primary keys and re-pulling the associated models from within your queue job. This often avoids unexpected serialization behavior.

When using Iron.io push queues, you should take extra precaution queueing Closures. The end-point that receives your queue messages should check for a token to verify that the request is actually from Iron.io. For example, your push queue end-point should be something like: `https://yourapp.com/queue/receive?token=SecretToken`. You may then check the value

of the secret token in your application before marshalling the queue request.

Running The Queue Listener

Laravel includes an Artisan task that will run new jobs as they are pushed onto the queue. You may run this task using the `queue:listen` command:

Starting The Queue Listener

```
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.

You may pass a comma-delimited list of queue connections to the `listen` command to set queue priorities:

```
php artisan queue:listen --queue=high,low
```

In this example, jobs on the `high-connection` will always be processed before moving onto jobs from the `low-connection`.

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.

Processing The First Job On The Queue

To process only the first job on the queue, you may use the `queue:work` command:

```
php artisan queue:work
```

Daemon Queue Worker

The `queue:work` also 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, but at the added complexity of needing to drain the queues of currently executing jobs during your deployments.

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` command supports most of the same options available to `queue:listen`. You may use the `php artisan help queue:work` command to view all of the available options.

Deploying With Daemon Queue Workers

The simplest way to deploy an application using daemon queue workers is to put the application in maintenance mode at the beginning of your deployment. This can be done using the `php artisan down` command. Once the application is in maintenance mode, Laravel will not accept any new jobs off of the queue, but will continue to process existing jobs.

The easiest way to restart your workers is to include the following command in your deployment script:

```
php artisan queue:restart
```

This command will instruct all queue workers to restart after they finish processing their current job.

> **Note:** This command relies on the cache system to schedule the restart. By default, APCu does not work for CLI commands. If you are using APCu, add `apc.enable_cli=1` to your APCu configuration.

Coding For Daemon Queue Workers

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 long-running daemon. You may use the `DB::reconnect` method to ensure you have a fresh connection.

Push Queues

Push queues allow you to utilize the powerful Laravel 4 queue facilities without running any daemons or background listeners. Currently, push queues are only supported by the Iron.io driver. Before getting started, create an Iron.io account, and add your Iron credentials to the `config/queue.php` configuration file.

Registering A Push Queue Subscriber

Next, you may use the `queue:subscribe` Artisan command to register a URL end-point that will receive newly pushed queue jobs:

```
php artisan queue:subscribe queue_name http://foo.com/queue/receive
```

Now, when you login to your Iron dashboard, you will see your new push queue, as well as the subscribed URL. You may subscribe as many URLs as you wish to a given queue. Next, create a route for your `queue/receive` end-point and return the response from the `Queue::marshal` method:

```
Route::post('queue/receive', function()
{
    return Queue::marshal();
});
```

The `marshal` method will take care of firing the correct job handler class. To fire jobs onto the push queue, just use the same `Queue::push` method used for conventional queues.

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 failed jobs table name 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
```

You can 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
```

If you would like to register an event that will be called when a queue job fails, you may use the `Queue::failing` method. This event is a great opportunity to notify your team via e-mail or HipChat.

```
Queue::failing(function($connection, $job, $data)
{
    //
});
```

You may also define a `failed` method directly on a queue job class, allowing you to perform job specific actions when a failure occurs:

```
public function failed()
{
    // Called when the job is failing...
}
```

Retrying Failed Jobs

To view all of your failed jobs, 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
```

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
```

Session

Configuration

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.

The session configuration is stored in `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 the majority of applications.

Before using Redis sessions with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer.

> **Note:** If you need all stored session data to be encrypted, set the `encrypt` configuration option to `true`.

Reserved Keys

The Laravel framework uses the `flash` session key internally, so you should not add an item to the session by that name.

Session Usage

Storing An Item In The Session

```
Session::put('key', 'value');
```

Push A Value Onto An Array Session Value

```
Session::push('user.teams', 'developers');
```

Retrieving An Item From The Session

```
$value = Session::get('key');
```

Retrieving An Item Or Returning A Default Value

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

$value = Session::get('key', function() { return 'default'; });
```

Retrieving An Item And Forgetting It

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

Retrieving All Data From The Session

```
$data = Session::all();
```

Determining If An Item Exists In The Session

```
if (Session::has('users'))
{
    //
}
```

Removing An Item From The Session

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

Removing All Items From The Session

```
Session::flush();
```

Regenerating The Session ID

```
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 `Session::flash` method:

```
Session::flash('key', 'value');
```

Reflashing The Current Flash Data For Another Request

```
Session::reflash();
```

Reflashing Only A Subset Of Flash Data

```
Session::keep(array('username', 'email'));
```

Database Sessions

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');
});
```

Of course, you may use the `session:table` Artisan command to generate this migration for you!

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

Session Drivers

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 will be stored in `app/storage/sessions`.
- `cookie` - sessions will be stored in secure, encrypted cookies.
- `database` - sessions will be stored in a database used by your application.
- `memcached` / `redis` - sessions will be stored in one of these fast, cached based stores.
- `array` - sessions will be stored in a simple PHP array and will not be persisted across requests.

> **Note:** The array driver is typically used for running unit tests, so no session data will be persisted.

Templates

Blade Templating

Blade is a simple, yet powerful templating engine provided with Laravel. Unlike controller layouts, Blade is driven by *template inheritance* and *sections*. All Blade templates should use the `.blade.php` extension.

Defining A Blade Layout

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

Using A Blade Layout

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

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@stop

@section('content')
    <p>This is my body content.</p>
@stop
```

Note that views which `extend` a Blade layout simply override sections from the layout. Content of the layout can be included in a child view using the `@@parent` directive in a section, allowing you to append to the contents of a layout section such as a sidebar or footer.

Sometimes, such as when you are not sure if a section has been defined, you may wish to pass a default value to the `@yield` directive. You may pass the default value as the second argument:

```
@yield('section', 'Default Content')
```

Other Blade Control Structures

Echoing Data

```
Hello, {{ $name }}.
```

```
The current UNIX timestamp is {{ time() }}.
```

Echoing Data After Checking For Existence

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. Basically, you want to do this:

```
{{ isset($name) ? $name : 'Default' }}
```

However, instead of writing a ternary statement, Blade allows you to use the following convenient short-cut:

```
{{ $name or 'Default' }}
```

Displaying Raw Text With Curly Braces

If you need to display a string that is wrapped in curly braces, you may escape the Blade behavior by prefixing your text with an `@` symbol:

```
@{{ This will not be processed by Blade }}
```

If you don't want the 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.

If Statements

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif

@unless (Auth::check())
    You are not signed in.
@endunless
```

Loops

```
@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

```
@include('view.name')
```

You may also pass an array of data to the included view:

```
@include('view.name', ['some' => 'data'])
```

Overwriting Sections

To overwrite a section entirely, you may use the `overwrite` statement:

```
@extends('list.item.container')

@section('list.item.content')
    <p>This is an item of type {{ $item->type }}</p>
@overwrite
```

Displaying Language Lines

```
@lang('language.line')

@choice('language.line', 1)
```

Comments

```
{{-- This comment will not be in the rendered HTML --}}
```

Extending Blade

Blade even allows you to define your own custom control structures. When a Blade file is compiled, each custom extension is called with the view contents, allowing you to do anything from simple `str_replace` manipulations to more complex regular expressions.

The Blade compiler comes with the helper methods `createMatcher` and `createPlainMatcher`, which generate the expression you need to build your own custom directives.

The `createPlainMatcher` method is used for directives with no arguments like `@endif` and `@stop`, while `createMatcher` is used for directives with arguments.

The following example creates a `@datetime($var)` directive which simply calls `->format()` on `$var`:

```
Blade::extend(function($view, $compiler)
{
    $pattern = $compiler->createMatcher('datetime');

    return preg_replace($pattern, '$1<?php echo $2->format(\'m/d/Y H:i\'); ?>', $view);
});
```

Testing

Introduction

Laravel is built with unit 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.

An example test file is provided in the `tests` directory. After installing a new Laravel application, simply run `phpunit` on the command line to run your tests.

Defining & Running Tests

To create a test case, simply create a new test file in the `tests` directory. The test class should extend `TestCase`. You may then define test methods as you normally would when using PHPUnit.

An Example Test Class

```
class FooTest extends TestCase {

    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

You may run all of the tests for your application by executing the `phpunit` command from your terminal.

> **Note:** If you define your own `setUp` method, be sure to call `parent::setUp`.

Test Environment

When running unit tests, Laravel will automatically set the configuration environment to `testing`. Also, Laravel includes configuration files for `session` and `cache` in the test environment. Both of these drivers are set to `array` while in the test environment, 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.

Calling Routes From Tests

Calling A Route From A Test

You may easily call one of your routes for a test using the `call` method:

```
$response = $this->call('GET', 'user/profile');

$response = $this->call($method, $uri, $parameters, $files, $server, $content);
```

You may then inspect the `Illuminate\Http\Response` object:

```
$this->assertEquals('Hello World', $response->getContent());
```

Calling A Controller From A Test

You may also call a controller from a test:

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

> **Note:** You do not need to specify the full controller namespace when using the `action` method. Only specify the portion of the class name that follows the `App\Http\Controllers` namespace.

The `getContent` method will return the evaluated string contents of the response. If your route returns a `View`, you may access it using the `original` property:

```
$view = $response->original;

$this->assertEquals('John', $view['name']);
```

To call a HTTPS route, you may use the `callSecure` method:

```
$response = $this->callSecure('GET', 'foo/bar');
```

Mocking Facades

When testing, you may often want to mock a call to a Laravel static facade. For example, consider the following controller action:

```
public function getIndex()
{
    Event::fire('foo', ['name' => 'Dayle']);

    return 'All done!';
}
```

We can mock the call to the `Event` class by using the `shouldReceive` method on the facade, which will return an instance of a Mockery mock.

Mocking A Facade

```
public function testGetIndex()
{
    Event::shouldReceive('fire')->once()->with('foo', ['name' => 'Dayle']);
```

```
    $this->call('GET', '/');
}
```

> **Note:** You should not mock the `Request` facade. Instead, pass the input you desire into the `call` method when running your test.

Framework Assertions

Laravel ships with several `assert` methods to make testing a little easier:

Asserting Responses Are OK

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertResponseOk();
}
```

Asserting Response Statuses

```
$this->assertResponseStatus(403);
```

Asserting Responses Are Redirects

```
$this->assertRedirectedTo('foo');

$this->assertRedirectedToRoute('route.name');

$this->assertRedirectedToAction('Controller@method');
```

Asserting A View Has Some Data

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertViewHas('name');
    $this->assertViewHas('age', $value);
}
```

Asserting The Session Has Some Data

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHas('name');
    $this->assertSessionHas('age', $value);
}
```

Asserting The Session Has Errors

```
public function testMethod()
{
```

```
    $this->call('GET', '/');

    $this->assertSessionHasErrors();

    // Asserting the session has errors for a given key...
    $this->assertSessionHasErrors('name');

    // Asserting the session has errors for several keys...
    $this->assertSessionHasErrors(array('name', 'age'));
}
```

Asserting Old Input Has Some Data

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertHasOldInput();
}
```

Helper Methods

The `TestCase` class contains several helper methods to make testing your application easier.

Setting And Flushing Sessions From Tests

```
$this->session(['foo' => 'bar']);

$this->flushSession();
```

Setting The Currently Authenticated User

You may set the currently authenticated user using the `be` method:

```
$user = new User(array('name' => 'John'));

$this->be($user);
```

You may re-seed your database from a test using the `seed` method:

Re-Seeding Database From Tests

```
$this->seed();

$this->seed($connection);
```

More information on creating seeds may be found in the migrations and seeding section of the documentation.

Refreshing The Application

As you may already know, you can access your Laravel `Application` / IoC Container via `$this->app` from any test method. This Application instance is refreshed for each test class. If you wish to manually force the Application to be refreshed for a given method, you may use the `refreshApplication` method from your test method. This will reset any extra bindings, such as mocks, that have been placed in the IoC container since the test case started running.

Validation

Basic Usage

Laravel ships with a simple, convenient facility for validating data and retrieving validation error messages via the `Validation` class.

Basic Validation Example

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => 'required|min:5')
);
```

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.

Using Arrays To Specify Rules

Multiple rules may be delimited using either a "pipe" character, or as separate elements of an array.

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => array('required', 'min:5'))
);
```

Validating Multiple Fields

```
$validator = Validator::make(
    array(
        'name' => 'Dayle',
        'password' => 'lamepassword',
        'email' => 'email@example.com'
    ),
    array(
        'name' => 'required',
        'password' => 'required|min:8',
        'email' => 'required|email|unique:users'
    )
);
```

Once a `Validator` instance has been created, the `fails` (or `passes`) method may be used to perform the validation.

```
if ($validator->fails())
{
```

```
    // The given data did not pass validation
}
```

If validation has failed, you may retrieve the error messages from the validator.

```
$messages = $validator->messages();
```

You may also access an array of the failed validation rules, without messages. To do so, use the `failed` method:

```
$failed = $validator->failed();
```

Validating Files

The `Validator` class provides several rules for validating files, such as `size`, `mimes`, and others. When validating files, you may simply pass them into the validator with your other data.

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())
{
    //
}
```

You may add as many `after` callbacks to a validator as needed.

Controller Validation

Of course, manually creating and checking a `Validator` instance each time you do validation is a headache. Don't worry, you have other options! The base `App\Http\Controllers\Controller` class included with Laravel uses a `ValidatesRequests` trait. This trait provides a single, convenient method for validating incoming HTTP requests. Here's what it looks like:

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    //
}
```

If validation passes, your code will keep executing normally. However, if validation fails, an `Illuminate\Contracts\Validation\ValidationException` will be thrown. This exception is automatically caught and a redirect is generated to the user's previous location. The validation errors are even automatically flashed to the session!

If the incoming request was an AJAX request, no redirect will be generated. Instead, an HTTP response with a 422 status code will be returned to the browser containing a JSON representation of the validation errors.

For example, here is the equivalent code written manually:

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $v = Validator::make($request->all(), [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    if ($v->fails())
    {
        return redirect()->back()->withErrors($v->errors());
    }

    //
}
```

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\Validation\Validator` class at the top of the file:

```
/**
 * {@inheritdoc}
 */
protected function formatValidationErrors(Validator $validator)
{
    return $validator->errors()->all();
}
```

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|max:255',
        'body' => 'required',
    ];
}
```

So, how are the validation rules executed? All you need to do is type-hint the request on your controller method:

```
/**
 * Store the incoming blog post.
 *
 * @param  StoreBlogPostRequest  $request
 * @return Response
 */
public function store(StoreBlogPostRequest $request)
{
    // The incoming request is valid...
}
```

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. It has already been validated!

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 `formatValidationErrors` on your base request (`App\Http\Requests\Request`). Don't forget to import the `Illuminate\Validation\Validator` class at the top of the file:

```
/**
 * {@inheritdoc}
 */
protected function formatErrors(Validator $validator)
{
    return $validator->errors()->all();
}
```

Working With Error Messages

After calling the `messages` method on a `Validator` instance, you will receive a `MessageBag` instance, which has a variety of convenient methods for working with error messages.

Retrieving The First Error Message For A Field

```
echo $messages->first('email');
```

Retrieving All Error Messages For A Field

```
foreach ($messages->get('email') as $message)
{
    //
}
```

Retrieving All Error Messages For All Fields

```
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>');
```

> **Note:** By default, messages are formatted using Bootstrap compatible syntax.

Retrieving All Error Messages With A Format

```
foreach ($messages->all('<li>:message</li>') as $message)
{
    //
}
```

Error Messages & Views

Once you have performed validation, you will need an easy way to get the error messages back to your views. This is conveniently handled by Laravel. Consider the following routes as an example:

```
Route::get('register', function()
{
    return View::make('user.register');
});

Route::post('register', function()
{
    $rules = array(...);

    $validator = Validator::make(Input::all(), $rules);

    if ($validator->fails())
    {
        return redirect('register')->withErrors($validator);
    }
});
```

Note that when validation fails, we pass the `Validator` instance to the Redirect using the `withErrors` method. This method will flash the error messages to the session so that they are available on the next request.

However, notice that we do 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 `MessageBag`.

So, after redirection, you may utilize the automatically bound `$errors` variable in your view:

```
<?php echo $errors->first('email'); ?>
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors. This will allow 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:

```
<?php echo $errors->login->first('email'); ?>
```

Available Validation Rules

Below is a list of all available validation rules and their function:

accepted

The field under validation must be *yes*, *on*, or *1*. 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 PHP `strtotime` function.

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 of type 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 *format* defined according to the `date_parse_from_format` PHP function.

different:*field*

The given *field* must be different than the field under validation.

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'
```

Passing `NULL` as a "where" clause value will add a check for a `NULL` database value:

```
'email' => 'exists:staff,email,deleted_at,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 have an integer value.

ip

The field under validation must be formatted as an IP address.

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'
```

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 have a numeric value.

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.

required*if:_field*,*value*,...

The field under validation must be present if the *field* 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:*value*

The field under validation must be a string type.

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.

Basic Usage Of Unique Rule

```
'email' => 'unique:users'
```

Specifying A Custom Column Name

```
'email' => 'unique:users,email_address'
```

Forcing A Unique Rule To Ignore A Given ID

```
'email' => 'unique:users,email_address,10'
```

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 formatted as an URL.

> **Note:** This function uses PHP's `filter_var` method.

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, array(
    '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 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, array(
    '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:

```
$v->sometimes(array('reason', 'cost'), 'required', function($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 as an object to access your input and files.

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.

Passing Custom Messages Into Validator

```
$messages = array(
    'required' => 'The :attribute field is required.',
);

$validator = Validator::make($input, $rules, $messages);
```

> *Note:* 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.

Other Validation Place-Holders

```
$messages = array(
    '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:

```
$messages = array(
    'email.required' => 'We need to know your e-mail address!',
);
```

Specifying Custom Messages In Language Files

In some cases, you may wish to specify your custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
'custom' => array(
    'email' => array(
        'required' => 'We need to know your e-mail address!',
    ),
),
```

Custom Validation Rules

Registering A Custom Validation Rule

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 `Validator::extend` method:

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

The custom validator Closure receives three arguments: the name of the `$attribute` being validated, the `$value` of the attribute, and an array of `$parameters` passed to the rule.

You may also pass a class and method to the `extend` method instead of a Closure:

```
Validator::extend('foo', 'FooValidator@validate');
```

Note that you will also need to define an error message for your custom rules. You can do so either using an inline custom message array or by adding an entry in the validation language file.

Extending The Validator Class

Instead of using Closure callbacks to extend the Validator, you may also extend the Validator class itself. To do so, write a Validator class that extends `Illuminate\Validation\Validator`. You may add validation methods to the class by prefixing them with `validate`:

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'foo';
    }
```

```
}
```

Registering A Custom Validator Resolver

Next, you need to register your custom Validator extension:

```
Validator::resolver(function($translator, $data, $rules, $messages)
{
    return new CustomValidator($translator, $data, $rules, $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, and adding a `replaceXXX` function to the validator.

```
protected function replaceFoo($message, $attribute, $rule, $parameters)
{
    return str_replace(':foo', $parameters[0], $message);
}
```

If you would like to add a custom message "replacer" without extending the `Validator` class, you may use the `Validator::replacer` method:

```
Validator::replacer('rule', function($message, $attribute, $rule, $parameters)
{
    //
});
```

Database

Basic Database Usage

Configuration

Laravel makes connecting with databases and running queries extremely simple. The database configuration file is `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.

Currently Laravel supports four database systems: MySQL, Postgres, SQLite, and SQL Server.

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 Queries

Once you have configured your database connection, you may run queries using the `DB` facade.

Running A Select Query

```
$results = DB::select('select * from users where id = ?', [1]);
```

The `select` method will always return an `array` of results.

Running An Insert Statement

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Running An Update Statement

```
DB::update('update users set votes = 100 where name = ?', ['John']);
```

Running A Delete Statement

```
DB::delete('delete from users');
```

> **Note:** The `update` and `delete` statements return the number of rows affected by the operation.

Running A General Statement

```
DB::statement('drop table users');
```

Listening For Query Events

You may listen for query events using the `DB::listen` method:

```
DB::listen(function($sql, $bindings, $time)
{
    //
});
```

Database Transactions

To run a set of operations within a database transaction, you may use the `transaction` method:

```
DB::transaction(function()
{
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

> **Note:** Any exception thrown within the `transaction` closure will cause the transaction to be rolled back automatically.

Sometimes you may need to begin a transaction yourself:

```
DB::beginTransaction();
```

You can rollback a transaction via the `rollback` method:

```
DB::rollback();
```

Lastly, you can commit a transaction via the `commit` method:

```
DB::commit();
```

Accessing Connections

When using multiple connections, you may access them via the `DB::connection` method:

```
$users = DB::connection('foo')->select(...);
```

You may also access the raw, underlying PDO instance:

```
$pdo = DB::connection()->getPdo();
```

Sometimes you may need to reconnect to a given database:

```
DB::reconnect('foo');
```

If you need to disconnect from the given database due to exceeding the underlying PDO instance's `max_connections` limit, use the `disconnect` method:

```
DB::disconnect('foo');
```

Query Logging

By default, Laravel keeps a log in memory of all queries that have been run for the current request. However, in some cases, such as when inserting a large number of rows, this can cause the application to use excess memory. To disable the log, you may use the `disableQueryLog` method:

```
DB::connection()->disableQueryLog();
```

To get an array of the executed queries, you may use the `getQueryLog` method:

```
$queries = DB::getQueryLog();
```

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 throughout to protect your application against SQL injection attacks. There is no need to clean strings being passed as bindings.

Selects

Retrieving All Rows From A Table

```
$users = DB::table('users')->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

Chunking Results From A Table

```
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)
{
    //

    return false;
});
```

Retrieving A Single Row From A Table

```
$user = DB::table('users')->where('name', 'John')->first();

var_dump($user->name);
```

Retrieving A Single Column From A Row

```
$name = DB::table('users')->where('name', 'John')->pluck('name');
```

Retrieving A List Of Column Values

```
$roles = DB::table('roles')->lists('title');
```

This method will return an array of role titles. You may also specify a custom key column for the returned array:

```
$roles = DB::table('roles')->lists('title', 'name');
```

Specifying A Select Clause

```
$users = DB::table('users')->select('name', 'email')->get();

$users = DB::table('users')->distinct()->get();

$users = DB::table('users')->select('name as user_name')->get();
```

Adding A Select Clause To An Existing Query

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Using Where Operators

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

Or Statements

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Using Where Between

```
$users = DB::table('users')
                    ->whereBetween('votes', array(1, 100))->get();
```

Using Where Not Between

```
$users = DB::table('users')
                    ->whereNotBetween('votes', array(1, 100))->get();
```

Using Where In With An Array

```
$users = DB::table('users')
                    ->whereIn('id', array(1, 2, 3))->get();

$users = DB::table('users')
                    ->whereNotIn('id', array(1, 2, 3))->get();
```

Using Where Null To Find Records With Unset Values

```
$users = DB::table('users')
                    ->whereNull('updated_at')->get();
```

Order By, Group By, And Having

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

Offset & Limit

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Joins

The query builder may also be used to write join statements. Take a look at the following examples:

Basic Join Statement

```
DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.id', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join Statement

```
DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
        ->get();
```

You may also specify more advanced join clauses:

```
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();
```

Advanced Wheres

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:

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function($query)
            {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

The query above will produce the following SQL:

```
select * from users where name = 'John' or (votes > 100 and title <> 'Admin')
```

Exists Statements

```
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
)
```

Aggregates

The query builder also provides a variety of aggregate methods, such as `count`, `max`, `min`, `avg`, and `sum`.

Using Aggregate Methods

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');

$price = DB::table('orders')->min('price');

$price = DB::table('orders')->avg('price');

$total = DB::table('users')->sum('votes');
```

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:

Using A Raw Expression

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

Inserts

Inserting Records Into A Table

```
DB::table('users')->insert(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

Inserting Records Into A Table With An Auto-Incrementing ID

If the table has an auto-incrementing id, use `insertGetId` to insert a record and retrieve the id:

```
$id = DB::table('users')->insertGetId(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

> **Note:** When using PostgreSQL the insertGetId method expects the auto-incrementing column to be named "id".

Inserting Multiple Records Into A Table

```
DB::table('users')->insert(array(
    array('email' => 'taylor@example.com', 'votes' => 0),
    array('email' => 'dayle@example.com', 'votes' => 0),
));
```

Updates

Updating Records In A Table

```
DB::table('users')
            ->where('id', 1)
            ->update(array('votes' => 1));
```

Incrementing or decrementing a value of a column

```
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:

```
DB::table('users')->increment('votes', 1, array('name' => 'John'));
```

Deletes

Deleting Records In A Table

```
DB::table('users')->where('votes', '<', 100)->delete();
```

Deleting All Records From A Table

```
DB::table('users')->delete();
```

Truncating A Table

```
DB::table('users')->truncate();
```

Unions

The query builder also provides a quick way to "union" two queries together:

```
$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`.

Pessimistic Locking

The query builder includes a few functions to help you do "pessimistic locking" on your SELECT statements.

To run the SELECT statement with a "shared lock", you may use the `sharedLock` method on a query:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

To "lock for update" on a SELECT statement, you may use the `lockForUpdate` method on a query:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

Eloquent ORM

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.

Before getting started, be sure to configure a database connection in `config/database.php`.

Basic Usage

To get started, 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`.

Defining An Eloquent Model

```
class User extends Model {}
```

You may also generate Eloquent models using the `make:model` command:

```
php artisan make:model User
```

Note that we did not tell Eloquent which table to use for our `User` model. The lower-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 `User` model stores records in the `users` table. You may specify a custom table by defining a `table` property on your model:

```
class User extends Model {
```

```
    protected $table = 'my_users';

}
```

> **Note:** Eloquent will also assume that each table has a primary key column named `id`. You may define a `primaryKey` property to override this convention. Likewise, you may define a `connection` property to override the name of the database connection that should be used when utilizing the model.

Once a model is defined, you are ready to start retrieving and creating records in your table. Note that you will need to place `updated_at` and `created_at` columns on your table by default. If you do not wish to have these columns automatically maintained, set the `$timestamps` property on your model to `false`.

Retrieving All Models

```
$users = User::all();
```

Retrieving A Record By Primary Key

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

var_dump($user->name);
```

> **Note:** All methods available on the query builder are also available when querying Eloquent models.

Retrieving A Model By Primary Key Or Throw An Exception

Sometimes you may wish to throw an exception if a model is not found, allowing you to catch the exceptions using an `App::error` handler and display a 404 page.

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

To register the error handler, listen for the `ModelNotFoundException`

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
});
```

Querying Using Eloquent Models

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

Eloquent Aggregates

Of course, you may also use the query builder aggregate functions.

```
$count = User::where('votes', '>', 100)->count();
```

If you are unable to generate the query you need via the fluent interface, feel free to use `whereRaw`:

```
$users = User::whereRaw('age > ? and votes = 100', array(25))->get();
```

Chunking Results

If you need to process a lot (thousands) of Eloquent records, using the `chunk` command will allow you to do without eating all of your RAM:

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

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 pulled from the database.

Specifying The Query Connection

You may also specify which database connection should be used when running an Eloquent query. Simply use the `on` method:

```
$user = User::on('connection-name')->find(1);
```

If you are using read / write connections, you may force the query to use the "write" connection with the following method:

```
$user = User::onWriteConnection()->find(1);
```

Mass Assignment

When creating a new model, you pass an array of attributes to the model constructor. These attributes are then assigned to the model via mass-assignment. This is convenient; however, can be a **serious** security concern when blindly passing user input into a model. If user input is blindly passed into a model, the user is free to modify **any** and **all** of the model's attributes. For this reason, all Eloquent models protect against mass-assignment by default.

To get started, set the `fillable` or `guarded` properties on your model.

Defining Fillable Attributes On A Model

The `fillable` property specifies which attributes should be mass-assignable. This can be set at the class or instance level.

```
class User extends Model {

    protected $fillable = array('first_name', 'last_name', 'email');

}
```

In this example, only the three listed attributes will be mass-assignable.

Defining Guarded Attributes On A Model

The inverse of `fillable` is `guarded`, and serves as a "black-list" instead of a "white-list":

```
class User extends Model {

    protected $guarded = array('id', 'password');

}
```

> **Note:** When using `guarded`, you should still never pass `Input::get()` or any raw array of user controlled input into a `save` or `update` method, as any column that is not guarded may be updated.

Blocking All Attributes From Mass Assignment

In the example above, the `id` and `password` attributes may **not** be mass assigned. All other attributes will be mass assignable. You may also block **all** attributes from mass assignment using the guard property:

```
protected $guarded = array('*');
```

Insert, Update, Delete

To create a new record in the database from a model, simply create a new model instance and call the `save` method.

Saving A New Model

```
$user = new User;

$user->name = 'John';

$user->save();
```

> **Note:** Typically, your Eloquent models will have auto-incrementing keys. However, if you wish to specify your own keys, set the `incrementing` property on your model to `false`.

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.

After saving or creating a new model that uses auto-incrementing IDs, you may retrieve the ID by accessing the object's `id` attribute:

```
$insertedId = $user->id;
```

Setting The Guarded Attributes On The Model

```
class User extends Model {

    protected $guarded = array('id', 'account_id');

}
```

Using The Model Create Method

```
// Create a new user in the database...
$user = User::create(array('name' => 'John'));

// Retrieve the user by the attributes, or create it if it doesn't exist...
$user = User::firstOrCreate(array('name' => 'John'));

// Retrieve the user by the attributes, or instantiate a new instance...
$user = User::firstOrNew(array('name' => 'John'));
```

Updating A Retrieved Model

To update a model, you may retrieve it, change an attribute, and use the `save` method:

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

$user->email = 'john@foo.com';

$user->save();
```

Saving A Model And Relationships

Sometimes you may wish to save not only a model, but also all of its relationships. To do so, you may use the `push` method:

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

You may also run updates as queries against a set of models:

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

> **Note:** No model events are fired when updating a set of models via the Eloquent query builder.

Deleting An Existing Model

To delete a model, simply call the `delete` method on the instance:

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

$user->delete();
```

Deleting An Existing Model By Key

```
User::destroy(1);

User::destroy(array(1, 2, 3));

User::destroy(1, 2, 3);
```

Of course, you may also run a delete query on a set of models:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

Updating Only The Model's Timestamps

If you wish to simply update the timestamps on a model, you may use the `touch` method:

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

Soft Deleting

When soft deleting a model, it is not actually removed from your database. Instead, a `deleted_at` timestamp is set on the record. To enable soft deletes for a model, apply the `SoftDeletes` to the model:

```
use Illuminate\Database\Eloquent\SoftDeletes;

class User extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

}
```

To add a `deleted_at` column to your table, you may use the `softDeletes` method from a migration:

```
$table->softDeletes();
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current timestamp. When querying a model that uses soft deletes, the "deleted" models will not be included in query results.

Forcing Soft Deleted Models Into Results

To force soft deleted models to appear in a result set, use the `withTrashed` method on the query:

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

The `withTrashed` method may be used on a defined relationship:

```
$user->posts()->withTrashed()->get();
```

If you wish to **only** receive soft deleted models in your results, you may use the `onlyTrashed` method:

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

To restore a soft deleted model into an active state, use the `restore` method:

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

You may also use the `restore` method on a query:

```
User::withTrashed()->where('account_id', 1)->restore();
```

Like with `withTrashed`, the `restore` method may also be used on relationships:

```
$user->posts()->restore();
```

If you wish to truly remove a model from the database, you may use the `forceDelete` method:

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

The `forceDelete` method also works on relationships:

```
$user->posts()->forceDelete();
```

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

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

Timestamps

By default, Eloquent will maintain the `created_at` and `updated_at` columns on your database table automatically. Simply add these `timestamp` columns to your table and Eloquent will take care of the rest. If you do not wish for Eloquent to maintain these columns, add the following property to your model:

Disabling Auto Timestamps

```
class User extends Model {

    protected $table = 'users';

    public $timestamps = false;

}
```

Providing A Custom Timestamp Format

If you wish to customize the format of your timestamps, you may override the `getDateFormat` method in your model:

```
class User extends Model {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

Query Scopes

Defining A Query Scope

Scopes allow you to easily re-use query logic in your models. To define a scope, simply prefix a model method with `scope`:

```
class User extends Model {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Utilizing A Query Scope

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. Just add your parameters to your scope function:

```
class User extends Model {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

Then pass the parameter into the scope call:

```
$users = User::ofType('member')->get();
```

Global Scopes

Sometimes you may wish to define a scope that applies to all queries performed on a model. In essence, this is how Eloquent's own "soft delete" feature works. Global scopes are defined using a combination of PHP traits and an implementation of `Illuminate\Database\Eloquent\ScopeInterface`.

First, let's define a trait. For this example, we'll use the `SoftDeletes` that ships with Laravel:

```
trait SoftDeletes {

    /**
     * Boot the soft deleting trait for a model.
     *
     * @return void
     */
    public static function bootSoftDeletes()
    {
        static::addGlobalScope(new SoftDeletingScope);
    }

}
```

If an Eloquent model uses a trait that has a method matching the `bootNameOfTrait` naming convention, that trait method will be called when the Eloquent model is booted, giving you an opportunity to register a global scope, or do anything else you want. A scope must implement `ScopeInterface`, which specifies two methods: `apply` and `remove`.

The `apply` method receives an `Illuminate\Database\Eloquent\Builder` query builder object, and is responsible for adding any additional `where` clauses that the scope wishes to add. The `remove` method also receives a `Builder` object and is responsible for reversing the action taken by `apply`. In other words, `remove` should remove the `where` clause (or any other clause) that was added. So, for our `SoftDeletingScope`, the methods look something like this:

```
/**
 * Apply the scope to a given Eloquent query builder.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @return void
 */
public function apply(Builder $builder)
{
    $model = $builder->getModel();

    $builder->whereNull($model->getQualifiedDeletedAtColumn());
}

/**
 * Remove the scope from the given Eloquent query builder.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @return void
 */
public function remove(Builder $builder)
{
    $column = $builder->getModel()->getQualifiedDeletedAtColumn();

    $query = $builder->getQuery();

    foreach ((array) $query->wheres as $key => $where)
    {
        // If the where clause is a soft delete date constraint, we will remove it from
        // the query and reset the keys on the wheres. This allows this developer to
        // include deleted model in a relationship result set that is lazy loaded.
        if ($this->isSoftDeleteConstraint($where, $column))
        {
            unset($query->wheres[$key]);

            $query->wheres = array_values($query->wheres);
        }
    }
}
```

Relationships

Of course, your database tables are probably 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. Laravel supports many types of relationships:

- One To One
- One To Many
- Many To Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

One To One

Defining A One To One Relation

A one-to-one relationship is a very basic relation. For example, a `User` model might have one `Phone`. We can define this

relation in Eloquent:

```
class User extends Model {

    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 it using Eloquent's dynamic properties:

```
$phone = User::find(1)->phone;
```

The SQL performed by this statement will be as follows:

```
select * from users where id = 1

select * from phones where user_id = 1
```

Take note that Eloquent assumes the foreign key of the relationship based on the model name. In this case, `Phone` model is assumed to use a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method. Furthermore, you may pass a third argument to the method to specify which local column that should be used for the association:

```
return $this->hasOne('App\Phone', 'foreign_key');

return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Defining The Inverse Of A Relation

To define the inverse of the relationship on the `Phone` model, we use the `belongsTo` method:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User');
    }

}
```

In the example above, Eloquent will look for a `user_id` column on the `phones` table. If you would like to define a different foreign key column, you may pass it as the second argument to the `belongsTo` method:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User', 'local_key');
    }

}
```

Additionally, you pass a third parameter which specifies the name of the associated column on the parent table:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User', 'local_key', 'parent_key');
    }

}
```

One To Many

An example of a one-to-many relation is a blog post that "has many" comments. We can model this relation like so:

```
class Post extends Model {

    public function comments()
    {
        return $this->hasMany('App\Comment');
    }

}
```

Now we can access the post's comments through the dynamic property:

```
$comments = Post::find(1)->comments;
```

If you need to add further constraints to which comments are retrieved, you may call the `comments` method and continue chaining conditions:

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

Again, you may override the conventional foreign key by passing a second argument to the `hasMany` method. And, like the `hasOne` relation, the local column may also be specified:

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Defining The Inverse Of A Relation

To define the inverse of the relationship on the `Comment` model, we use the `belongsTo` method:

```
class Comment extends Model {

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

Many To Many

Many-to-many relations are a more complicated relationship type. 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". Three database tables are needed for this relationship: `users`, `roles`, and `role_user`. The `role_user` table is derived from the

alphabetical order of the related model names, and should have `user_id` and `role_id` columns.

We can define a many-to-many relation using the `belongsToMany` method:

```
class User extends Model {

    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }

}
```

Now, we can retrieve the roles through the `User` model:

```
$roles = User::find(1)->roles;
```

If you would like to use an unconventional table name for your pivot table, you may pass it as the second argument to the `belongsToMany` method:

```
return $this->belongsToMany('App\Role', 'user_roles');
```

You may also override the conventional associated keys:

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'foo_id');
```

Of course, you may also define the inverse of the relationship on the `Role` model:

```
class Role extends Model {

    public function users()
    {
        return $this->belongsToMany('App\User');
    }

}
```

Has Many Through

The "has many through" relation provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a `Country` model might have many `Post` through a `User` model. The tables for this relationship would look like this:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Even though the `posts` table does not contain a `country_id` column, the `hasManyThrough` relation will allow us to access a

country's posts via `$country->posts`. Let's define the relationship:

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'User');
    }

}
```

If you would like to manually specify the keys of the relationship, you may pass them as the third and fourth arguments to the method:

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'User', 'country_id', 'user_id');
    }

}
```

Polymorphic Relations

Polymorphic relations allow a model to belong to more than one other model, on a single association. For example, you might have a photo model that belongs to either a staff model or an order model. We would define this relation like so:

```
class Photo extends Model {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}

class Order extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}
```

Retrieving A Polymorphic Relation

Now, we can retrieve the photos for either a staff member or an order:

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

Retrieving The Owner Of A Polymorphic Relation

However, the true "polymorphic" magic is when you access the staff or order from the `Photo` model:

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

The `imageable` relation on the `Photo` model will return either a `Staff` or `Order` instance, depending on which type of model owns the photo.

Polymorphic Relation Table Structure

To help understand how this works, let's explore the database structure for a polymorphic relation:

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

The key fields to notice here are the `imageable_id` and `imageable_type` on the `photos` table. The ID will contain the ID value of, in this example, the owning staff or order, while the type will contain the class name of the owning model. This is what allows the ORM to determine which type of owning model to return when accessing the `imageable` relation.

Many To Many Polymorphic Relations

Polymorphic Many To Many Relation Table Structure

In addition to traditional polymorphic relations, you may also specify many-to-many polymorphic relations. For example, a blog `Post` and `Video` model could share a polymorphic relation to a `Tag` model. 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
```

Next, we're ready to setup the relationships on the model. The `Post` and `Video` model will both have a `morphToMany` relationship via a `tags` method:

```
class Post extends Model {
```

```
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }

}
```

The `Tag` model may define a method for each of its relationships:

```
class Tag extends Model {

    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }

}
```

Querying Relations

Querying Relations When Selecting

When accessing the records for a model, you may wish to limit your results based on the existence of a relationship. For example, you wish to pull all blog posts that have at least one comment. To do so, you may use the `has` method:

```
$posts = Post::has('comments')->get();
```

You may also specify an operator and a count:

```
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may also be constructed using "dot" notation:

```
$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:

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');

})->get();
```

Dynamic Properties

Eloquent allows you to access your relations via dynamic properties. Eloquent will automatically load the relationship for you, and is even smart enough to know whether to call the `get` (for one-to-many relationships) or `first` (for one-to-one relationships) method. It will then be accessible via a dynamic property by the same name as the relation. For example, with the following model `$phone`:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User');
    }

}

$phone = Phone::find(1);
```

Instead of echoing the user's email like this:

```
echo $phone->user()->first()->email;
```

It may be shortened to simply:

```
echo $phone->user->email;
```

> **Note:** Relationships that return many results will return an instance of the `Illuminate\Database\Eloquent\Collection` class.

Eager Loading

Eager loading exists to alleviate the N + 1 query problem. For example, consider a `Book` model that is related to `Author`. The relationship is defined like so:

```
class Book extends Model {

    public function author()
    {
        return $this->belongsTo('App\Author');
    }

}
```

Now, consider the following code:

```
foreach (Book::all() 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.

Thankfully, we can use eager loading to drastically reduce the number of queries. The relationships that should be eager loaded may be specified via the `with` method:

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

In the loop above, only two queries will be executed:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Wise use of eager loading can drastically increase the performance of your application.

Of course, you may eager load multiple relationships at one time:

```
$books = Book::with('author', 'publisher')->get();
```

You may even eager load nested relationships:

```
$books = Book::with('author.contacts')->get();
```

In the example above, the `author` relationship will be eager loaded, and the author's `contacts` relation will also be loaded.

Eager Load Constraints

Sometimes you may wish to eager load a relationship, but also specify a condition for the eager load. Here's an example:

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');

}))->get();
```

In this example, we're eager loading the user's posts, but only if the post's title column contains the word "first".

Of course, eager loading Closures aren't limited to "constraints". You may also apply orders:

```
$users = User::with(array('posts' => function($query)
{
    $query->orderBy('created_at', 'desc');

}))->get();
```

Lazy Eager Loading

It is also possible to eagerly load related models directly from an already existing model collection. This may be useful when dynamically deciding whether to load related models or not, or in combination with caching.

```
$books = Book::all();

$books->load('author', 'publisher');
```

Inserting Related Models

Attaching A Related Model

You will often need to insert new related models. For example, you may wish to insert a new comment for a post. Instead of manually setting the `post_id` foreign key on the model, you may insert the new comment from its parent `Post` model directly:

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

In this example, the `post_id` field will automatically be set on the inserted comment.

If you need to save multiple related models:

```
$comments = array(
    new Comment(array('message' => 'A new comment.')),
    new Comment(array('message' => 'Another comment.')),
    new Comment(array('message' => 'The latest comment.'))
);

$post = Post::find(1);

$post->comments()->saveMany($comments);
```

Associating Models (Belongs To)

When updating a `belongsTo` relationship, you may use the `associate` method. This method will set the foreign key on the child model:

```
$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

Inserting Related Models (Many To Many)

You may also insert related models when working with many-to-many relations. Let's continue using our `User` and `Role` models as examples. We can easily attach new roles to a user using the `attach` method:

Attaching Many To Many Models

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

$user->roles()->attach(1);
```

You may also pass an array of attributes that should be stored on the pivot table for the relation:

```
$user->roles()->attach(1, array('expires' => $expires));
```

Of course, the opposite of `attach` is `detach`:

```
$user->roles()->detach(1);
```

Both `attach` and `detach` also take arrays of IDs as input:

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

$user->roles()->detach([1, 2, 3]);
```

```
$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

Using Sync To Attach Many To Many Models

You may also use the `sync` method to attach related models. The `sync` method accepts an array of IDs to place on the pivot table. After this operation is complete, only the IDs in the array will be on the intermediate table for the model:

```
$user->roles()->sync(array(1, 2, 3));
```

Adding Pivot Data When Syncing

You may also associate other pivot table values with the given IDs:

```
$user->roles()->sync(array(1 => array('expires' => true)));
```

Sometimes you may wish to create a new related model and attach it in a single command. For this operation, you may use the `save` method:

```
$role = new Role(array('name' => 'Editor'));

User::find(1)->roles()->save($role);
```

In this example, the new `Role` model will be saved and attached to the user model. You may also pass an array of attributes to place on the joining table for this operation:

```
User::find(1)->roles()->save($role, array('expires' => $expires));
```

Touching Parent Timestamps

When a model `belongsTo` another model, such as a `Comment` which belongs to a `Post`, it is often 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:

```
class Comment extends Model {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated:

```
$comment = Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

Working With Pivot Tables

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 `pivot` table on the models:

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

foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

Notice that each `Role` model we retrieve is automatically assigned a `pivot` attribute. This attribute contains a model representing the intermediate table, and may be used as any other Eloquent model.

By default, only the 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('foo', 'bar');
```

Now the `foo` and `bar` attributes will be accessible on our `pivot` object for the `Role` model.

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();
```

Deleting Records On A Pivot Table

To delete all records on the pivot table for a model, you may use the `detach` method:

```
User::find(1)->roles()->detach();
```

Note that this operation does not delete records from the `roles` table, but only from the pivot table.

Updating A Record On A Pivot Table

Sometimes you may need to update your pivot table, but not detach it. If you wish to update your pivot table in place you may use `updateExistingPivot` method like so:

```
User::find(1)->roles()->updateExistingPivot($roleId, $attributes);
```

Defining A Custom Pivot Model

Laravel also allows you to define a custom Pivot model. To define a custom model, first create your own "Base" model class that extends `Eloquent`. In your other Eloquent models, extend this custom base model instead of the default `Eloquent` base. In your base model, add the following function that returns an instance of your custom Pivot model:

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
```

```
}
```

Collections

All multi-result sets returned by Eloquent, either via the `get` method or a `relationship`, will return a collection object. This object implements the `IteratorAggregate` PHP interface so it can be iterated over like an array. However, this object also has a variety of other helpful methods for working with result sets.

Checking If A Collection Contains A Key

For example, we may determine if a result set contains a given primary key using the `contains` method:

```
$roles = User::find(1)->roles;

if ($roles->contains(2))
{
    //
}
```

Collections may also be converted to an array or JSON:

```
$roles = User::find(1)->roles->toArray();

$roles = User::find(1)->roles->toJson();
```

If a collection is cast to a string, it will be returned as JSON:

```
$roles = (string) User::find(1)->roles;
```

Iterating Collections

Eloquent collections also contain a few helpful methods for looping and filtering the items they contain:

```
$roles = $user->roles->each(function($role)
{
    //
});
```

Filtering Collections

When filtering collections, the callback provided will be used as callback for array_filter.

```
$users = $users->filter(function($user)
{
    return $user->isAdmin();
});
```

> **Note:** When filtering a collection and converting it to JSON, try calling the `values` function first to reset the array's keys.

Applying A Callback To Each Collection Object

```
$roles = User::find(1)->roles;
```

```
$roles->each(function($role)
{
    //
});
```

Sorting A Collection By A Value

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});
```

Sorting A Collection By A Value

```
$roles = $roles->sortBy('created_at');
```

Returning A Custom Collection Type

Sometimes, you may wish to return a custom Collection object with your own added methods. You may specify this on your Eloquent model by overriding the `newCollection` method:

```
class User extends Model {

    public function newCollection(array $models = array())
    {
        return new CustomCollection($models);
    }

}
```

Accessors & Mutators

Defining An Accessor

Eloquent provides a convenient way to transform your model attributes when getting or setting them. Simply define a `getFooAttribute` method on your model to declare an accessor. Keep in mind that the methods should follow camel-casing, even though your database columns are snake-case:

```
class User extends Model {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

In the example above, the `first_name` column has an accessor. Note that the value of the attribute is passed to the accessor.

Defining A Mutator

Mutators are declared in a similar fashion:

```
class User extends Model {
```

```
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

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 `getDates` method of the model:

```
public function getDates()
{
    return array('created_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.

To totally disable date mutations, simply return an empty array from the `getDates` method:

```
public function getDates()
{
    return array();
}
```

Attribute Casting

If you have some attributes that you want to always convert to another data-type, you may add the attribute to the `casts` property of your model. Otherwise, you will have to define a mutator for each of the attributes, which can be time consuming. Here is an example of using the `casts` property:

```
/**
 * 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. Other supported cast types are: `integer` , `real` , `float` , `double` , `string` , `boolean` , and `array` .

The `array` cast is particularly useful for working with columns that are stored as serialized JSON. For example, if your database has a TEXT type field 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:

```
/**
 * The attributes that should be casted to native types.
 *
 * @var array
```

```
 */
protected $casts = [
    'options' => 'array',
];
```

Now, when you utilize the Eloquent model:

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

// $options is an array...
$options = $user->options;

// options is automatically serialized back to JSON...
$user->options = ['foo' => 'bar'];
```

Model 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`.

Whenever a new item is saved for the first time, the `creating` and `created` events will fire. If an item is not new and the `save` method is called, the `updating` / `updated` events will fire. In both cases, the `saving` / `saved` events will fire.

Cancelling Save Operations Via Events

If `false` is returned from the `creating`, `updating`, `saving`, or `deleting` events, the action will be cancelled:

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

Where To Register Event Listeners

Your `EventServiceProvider` serves as a convenient place to register your model event bindings. For example:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    User::creating(function($user)
    {
        //
    });
}
```

Model Observers

To consolidate the handling of model events, you may register a model observer. An observer class may have methods that correspond to the various model events. For example, `creating`, `updating`, `saving` methods may be on an observer, in addition to any other model event name.

So, for example, a model observer might look like this:

```
class UserObserver {

    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
        //
    }

}
```

You may register an observer instance using the `observe` method:

```
User::observe(new UserObserver);
```

Converting To Arrays / JSON

Converting A Model To An Array

When building JSON APIs, you may often need to convert your models and relationships to arrays or JSON. So, Eloquent includes methods for doing so. To convert a model and its loaded relationship to an array, you may use the `toArray` method:

```
$user = User::with('roles')->first();

return $user->toArray();
```

Note that entire collections of models may also be converted to arrays:

```
return User::all()->toArray();
```

Converting A Model To JSON

To convert a model to JSON, you may use the `toJson` method:

```
return User::find(1)->toJson();
```

Returning A Model From A Route

Note that when a model or collection is cast to a string, it will be converted to JSON, meaning you can return Eloquent objects directly from your application's routes!

```
Route::get('users', function()
{
    return User::all();
});
```

Hiding Attributes From Array Or JSON Conversion

Sometimes you may wish to limit the attributes that are included in your model's array or JSON form, such as passwords.

To do so, add a `hidden` property definition to your model:

```
class User extends Model {

    protected $hidden = array('password');

}
```

> **Note:** When hiding relationships, use the relationship's **method** name, not the dynamic accessor name.

Alternatively, you may use the `visible` property to define a white-list:

```
protected $visible = array('first_name', 'last_name');
```

Occasionally, you may need to add array attributes that do not have a corresponding column in your database. To do so, simply define an accessor for the value:

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

Once you have created the accessor, just add the value to the `appends` property on the model:

```
protected $appends = array('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 respect the `visible` and `hidden` configuration on the model.

Schema Builder

Introduction

The Laravel `Schema` class provides a database agnostic way of manipulating tables. It works well with all of the databases supported by Laravel, and has a unified API across all of these systems.

Creating & Dropping Tables

To create a new database table, the `Schema::create` method is used:

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

The first argument passed to the `create` method is the name of the table, and the second is a `Closure` which will receive a `Blueprint` object which may be used to define the new table.

To rename an existing database table, the `rename` method may be used:

```
Schema::rename($from, $to);
```

To specify which connection the schema operation should take place on, use the `Schema::connection` method:

```
Schema::connection('foo')->create('users', function($table)
{
    $table->increments('id');
});
```

To drop a table, you may use the `Schema::drop` method:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Adding Columns

To update an existing table, we will use the `Schema::table` method:

```
Schema::table('users', function($table)
{
    $table->string('email');
});
```

The table builder contains a variety of column types that you may use when building your tables:

Command	Description
`$table->bigIncrements('id');`	Incrementing ID using a "big integer" equivalent.
`$table->bigInteger('votes');`	BIGINT equivalent to the table
`$table->binary('data');`	BLOB equivalent to the table
`$table->boolean('confirmed');`	BOOLEAN equivalent to the table
`$table->char('name', 4);`	CHAR equivalent with a length
`$table->date('created_at');`	DATE equivalent to the table
`$table->dateTime('created_at');`	DATETIME equivalent to the table
`$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', array('foo', 'bar'));`	ENUM equivalent to the table
`$table->float('amount');`	FLOAT equivalent to the table
`$table->increments('id');`	Incrementing ID to the table (primary key).
`$table->integer('votes');`	INTEGER equivalent to the table
`$table->json('options');`	JSON equivalent to the table
`$table->longText('description');`	LONGTEXT equivalent to the table
`$table->mediumInteger('numbers');`	MEDIUMINT equivalent to the table
`$table->mediumText('description');`	MEDIUMTEXT equivalent to the table
`$table->morphs('taggable');`	Adds INTEGER `taggable_id` and STRING `taggable_type`
`$table->nullableTimestamps();`	Same as `timestamps()`, except allows NULLs
`$table->smallInteger('votes');`	SMALLINT equivalent to the table
`$table->tinyInteger('numbers');`	TINYINT equivalent to the table
`$table->softDeletes();`	Adds **deleted_at** column for soft deletes
`$table->string('email');`	VARCHAR equivalent column
`$table->string('name', 100);`	VARCHAR equivalent with a length
`$table->text('description');`	TEXT equivalent to the table
`$table->time('sunrise');`	TIME equivalent to the table
`$table->timestamp('added_on');`	TIMESTAMP equivalent to the table
`$table->timestamps();`	Adds **created_at** and **updated_at** columns
`$table->rememberToken();`	Adds `remember_token` as VARCHAR(100) NULL
`->nullable()`	Designate that the column allows NULL values
`->default($value)`	Declare a default value for a column

`->unsigned()`	Set INTEGER to UNSIGNED

Using After On MySQL

If you are using the MySQL database, you may use the `after` method to specify the order of columns:

```
$table->string('name')->after('email');
```

Changing Columns

Sometimes you may need to modify an existing column. For example, you may wish to increase the size of a string column. The `change` method makes it easy! For example, 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 `enum` column types is not supported.

Dropping Columns

To drop a column, you may use the `dropColumn` method on the Schema builder. Before dropping a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file.

Dropping A Column From A Database Table

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes');
});
```

Dropping Multiple Columns From A Database Table

```
Schema::table('users', function($table)
{
    $table->dropColumn(array('votes', 'avatar', 'location'));
});
```

Checking Existence

Checking For Existence Of Table

You may easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users'))
{
    //
}
```

Checking For Existence Of Columns

```
if (Schema::hasColumn('users', 'email'))
{
    //
}
```

Adding Indexes

The schema builder supports several types of indexes. There are two ways to add them. First, you may fluently define them on a column definition, or you may add them separately:

```
$table->string('email')->unique();
```

Or, you may choose to add the indexes on separate lines. Below is a list of all available index types:

Command	Description
`$table->primary('id');`	Adding a primary key
`$table->primary(array('first', 'last'));`	Adding composite keys
`$table->unique('email');`	Adding a unique index
`$table->index('state');`	Adding a basic index

Foreign Keys

Laravel also provides support for adding foreign key constraints to your tables:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')->on('users');
```

In this example, we are stating that the `user_id` column references the `id` column on the `users` table. Make sure to create the foreign key column first!

You may also specify options for the "on delete" and "on update" actions of the constraint:

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

To drop a foreign key, you may use the `dropForeign` method. A similar naming convention is used for foreign keys as is used for other indexes:

```
$table->dropForeign('posts_user_id_foreign');
```

> **Note:** When creating a foreign key that references an incrementing integer, remember to always make the foreign key column `unsigned`.

Dropping Indexes

To drop an index you must specify the index's name. Laravel assigns a reasonable name to the indexes by default. Simply concatenate the table name, the names of the column in the index, and the index type. Here are some examples:

Command	Description
`$table->dropPrimary('users_id_primary');`	Dropping a primary key from the "users" table
`$table->dropUnique('users_email_unique');`	Dropping a unique index from the "users" table
`$table->dropIndex('geo_state_index');`	Dropping a basic index from the "geo" table

Dropping Timestamps & SoftDeletes

To drop the `timestamps`, `nullableTimestamps` or `softDeletes` column types, you may use the following methods:

Command	Description
`$table->dropTimestamps();`	Dropping the **created_at** and **updated_at** columns from the table
`$table->dropSoftDeletes();`	Dropping **deleted_at** column from the table

Storage Engines

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->string('email');
});
```

Migrations & Seeding

Introduction

Migrations are a type of version control for your database. They allow a team to modify the database schema and stay up to date on the current schema state. Migrations are typically paired with the Schema Builder to easily manage your application's schema.

Creating Migrations

To create a migration, you may use the `make:migration` command on the Artisan CLI:

```
php artisan make:migration create_users_table
```

The migration will be placed in your `database/migrations` folder, and will contain a timestamp which allows the framework 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:

```
php artisan make:migration add_votes_to_user_table --table=users

php artisan make:migration create_users_table --create=users
```

Running Migrations

Running All Outstanding Migrations

```
php artisan migrate
```

> **Note:** If you receive a "class not found" error when running migrations, try running the `composer dump-autoload` command.

Forcing Migrations 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 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

Rollback The Last Migration Operation

```
php artisan migrate:rollback
```

Rollback all migrations

```
php artisan migrate:reset
```

Rollback all migrations and run them all again

```
php artisan migrate:refresh

php artisan migrate:refresh --seed
```

Database Seeding

Laravel also includes a simple way to seed 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 `UserTableSeeder`, 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.

Example Database Seed Class

```
class DatabaseSeeder extends Seeder {

    public function run()
    {
        $this->call('UserTableSeeder');

        $this->command->info('User table seeded!');
    }

}

class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        User::create(array('email' => 'foo@bar.com'));
    }

}
```

To seed your database, you may use the `db:seed` command on the Artisan CLI:

```
php artisan db:seed
```

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 --class=UserTableSeeder
```

You may also seed your database using the `migrate:refresh` command, which will also rollback and re-run all of your migrations:

```
php artisan migrate:refresh --seed
```

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.

> **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.

Configuration

The Redis configuration for your application is stored in the `config/database.php` file. Within this file, you will see a `redis` array containing the Redis servers used by your application:

```
'redis' => [

    'cluster' => true,

    'default' => ['host' => '127.0.0.1', 'port' => 6379],

],
```

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.

If your Redis server requires authentication, you may supply a password by adding a `password` key / value pair to your Redis server configuration array.

Usage

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');
```

Once you have an instance of the Redis client, we may issue any of the Redis commands to the instance. Laravel uses magic methods to pass the commands to the Redis server:

```
$redis->set('name', 'Taylor');

$name = $redis->get('name');

$values = $redis->lrange('names', 5, 10);
```

Notice the arguments to the command are simply passed into the magic method. Of course, you are not required to use the magic methods, you may also pass commands to the server using the `command` method:

```
$values = $redis->command('lrange', array(5, 10));
```

When you are simply executing commands against the default connection, just use static magic methods on the `Redis` class:

```
Redis::set('name', 'Taylor');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

> **Note:** Redis cache and session drivers are included with Laravel.

Pipelining

Pipelining should be used when you need to send many commands to the server in one operation. To get started, use the `pipeline` command:

Piping Many Commands To Your Servers

```
Redis::pipeline(function($pipe)
{
    for ($i = 0; $i < 1000; $i++)
    {
        $pipe->set("key:$i", $i);
    }
});
```

Artisan CLI

Artisan CLI

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.

Usage

Listing All Available Commands

To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

Viewing The Help Screen For A Command

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
```

Specifying The Configuration Environment

You may specify the configuration environment that should be used while running a command using the `--env` switch:

```
php artisan migrate --env=local
```

Displaying Your Current Laravel Version

You may also view the current version of your Laravel installation using the `--version` option:

```
php artisan --version
```

Calling Commands Outside Of CLI

Sometimes you may wish to execute an Artisan command outside of the CLI. For example, you may wish to fire an Artisan command from an HTTP route. Just use the `Artisan` facade:

```
Route::get('/foo', function()
{
    $exitCode = Artisan::call('command:name', ['--option' => 'foo']);
```

```
    //
});
```

You may even queue Artisan commands so they are processed in the background by your queue workers:

```
Route::get('/foo', function()
{
    Artisan::queue('command:name', ['--option' => 'foo']);

    //
});
```

Scheduling Artisan Commands

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.

Your command schedule is stored in the `app/Console/Kernel.php` file. Within this class you will see a `schedule` method. To help you get started, a simple example is included with the method. You are free to add as many scheduled jobs as you wish to the `Schedule` object. The only Cron entry you need to add to your server is this:

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

This Cron will call the Laravel command scheduler every minute. Then, Laravel evaluates your scheduled jobs and runs the jobs that are due. It couldn't be easier!

More Scheduling Examples

Let's look at a few more scheduling examples:

Scheduling Closures

```
$schedule->call(function()
{
    // Do some task...

})->hourly();
```

Scheduling Terminal Commands

```
$schedule->exec('composer self-update')->daily();
```

Manual Cron Expression

```
$schedule->command('foo')->cron('* * * * *');
```

Frequent Jobs

```
$schedule->command('foo')->everyFiveMinutes();
```

```
$schedule->command('foo')->everyTenMinutes();

$schedule->command('foo')->everyThirtyMinutes();
```

Daily Jobs

```
$schedule->command('foo')->daily();
```

Daily Jobs At A Specific Time (24 Hour Time)

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

Twice Daily Jobs

```
$schedule->command('foo')->twiceDaily();
```

Job That Runs Every Weekday

```
$schedule->command('foo')->weekdays();
```

Weekly Jobs

```
$schedule->command('foo')->weekly();

// Schedule weekly job for specific day (0-6) and time...
$schedule->command('foo')->weeklyOn(1, '8:00');
```

Monthly Jobs

```
$schedule->command('foo')->monthly();
```

Limit The Environment The Jobs Should Run In

```
$schedule->command('foo')->monthly()->environments('production');
```

Indicate The Job Should Run Even When Application Is In Maintenance Mode

```
$schedule->command('foo')->monthly()->evenInMaintenanceMode();
```

Only Allow Job To Run When Callback Is True

```
$schedule->command('foo')->monthly()->when(function()
{
    return true;
});
```

Artisan Development

Introduction

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.

Building A Command

Generating The Class

To create a new command, you may use the `make:console` Artisan command, which will generate a command stub to help you get started:

Generate A New Command Class

```
php artisan make:console FooCommand
```

The command above would generate a class at `app/Console/FooCommand.php`.

When creating the command, the `--command` option may be used to assign the terminal command name:

```
php artisan make:console AssignUsers --command=users:assign
```

Writing The Command

Once your command is generated, you should fill out the `name` and `description` properties of the class, which will be used when displaying your command on the `list` screen.

The `fire` method will be called when your command is executed. You may place any command logic in this method.

Arguments & Options

The `getArguments` and `getOptions` methods are where you may define any arguments or options your command receives. Both of these methods return an array of commands, which are described by a list of array options.

When defining `arguments`, the array definition values represent the following:

```
array($name, $mode, $description, $defaultValue)
```

The argument `mode` may be any of the following: `InputArgument::REQUIRED` or `InputArgument::OPTIONAL`.

When defining `options`, the array definition values represent the following:

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

For options, the argument `mode` may be: `InputOption::VALUE_REQUIRED`, `InputOption::VALUE_OPTIONAL`, `InputOption::VALUE_IS_ARRAY`, `InputOption::VALUE_NONE`.

The `VALUE_IS_ARRAY` mode indicates that the switch may be used multiple times when calling the command:

```
php artisan foo --option=bar --option=baz
```

The `VALUE_NONE` option indicates that the option is simply used as a "switch":

```
php artisan foo --option
```

Retrieving Input

While your command is executing, you will obviously need to access the values for the arguments and options accepted by your application. To do so, you may use the `argument` and `option` methods:

Retrieving The Value Of A Command Argument

```
$value = $this->argument('name');
```

Retrieving All Arguments

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

Retrieving The Value Of A Command Option

```
$value = $this->option('name');
```

Retrieving All Options

```
$options = $this->option();
```

Writing Output

To send output to the console, you may use the `info`, `comment`, `question` and `error` methods. Each of these methods will use the appropriate ANSI colors for their purpose.

Sending Information To The Console

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

Sending An Error Message To The Console

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

Asking Questions

You may also use the `ask` and `confirm` methods to prompt the user for input:

Asking The User For Input

```
$name = $this->ask('What is your name?');
```

Asking The User For Secret Input

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

Asking The User For Confirmation

```
if ($this->confirm('Do you wish to continue? [yes|no]'))
{
    //
}
```

You may also specify a default value to the `confirm` method, which should be `true` or `false`:

```
$this->confirm($question, true);
```

Calling Other Commands

Sometimes you may wish to call other commands from your command. You may do so using the `call` method:

```
$this->call('command:name', ['argument' => 'foo', '--option' => 'bar']);
```

Registering Commands

Registering An Artisan Command

Once your command is finished, you need to register it with Artisan so it will be available for use. This is typically done in the `app/Console/Kernel.php` file. Within this file, you will find a list of commands in the `commands` property. To register your command, simply add it to this list. When Artisan boots, all the commands listed in this property will be resolved by the IoC container and registered with Artisan.