LARAVEL 4

THE PHP FRAMEWORK FOR WEB ARTISANS.

JUST GOT FOUR TIMES BETTER.

Introduction

Laravel Philosophy

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable, creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as authentication, routing, sessions, and caching.

Laravel aims to make the development process a pleasing one for the developer without sacrificing application functionality. Happy developers make the best code. To this end, we've attempted to combine the very best of what we have seen in other web frameworks, including frameworks implemented in other languages, such as Ruby on Rails, ASP.NET MVC, and Sinatra.

Laravel is accessible, yet powerful, providing powerful tools needed for large, robust applications. A superb inversion of control container, expressive migration system, and tightly integrated unit testing support give you the tools you need to build any application with which you are tasked.

Learning Laravel

One of the best ways to learn Laravel is to read through the entirety of its documentation. This guide details all aspects of the framework and how to apply them to your application.

In addition to this guide, you may wish to check out some Laravel books. These community written books serve as a good supplemental resource for learning about the framework:

- Code Bright by Dayle Rees
- Laravel Testing Decoded by Jeffrey Way
- Laravel: From Apprentice To Artisan by Taylor Otwell

Development Team

Laravel was created by Taylor Otwell, who continues to lead development of the framework. Other prominent community members and contributors include

Dayle Rees, Shawn McCool, Jeffrey Way, Jason Lewis, Ben Corlett, Franz Liedke, Dries Vints, Mior Muhammad Zaki, and Phil Sturgeon.

Framework Sponsors

The following organizations have made financial contributions to the development of the Laravel framework:

- UserScape
- Cartalyst
- Elli Davis - Toronto Realtor
- Jay Banks - Vancouver Lofts & Condos
- Julie Kinnear - Toronto MLS
- Jamie Sarner - Toronto Real Estate

Laravel Quickstart

Installation

To install the Laravel framework, you may issue the following command from your terminal:

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

Or, you may also download a copy of the repository from Github. Next, after installing Composer, run the `composer install` command in the root of your project directory. This command will download and install the framework's dependencies.

After installing the framework, take a glance around the project to familiarize yourself with the directory structure. The `app` directory contains folders such as `views`, `controllers`, and `models`. Most of your application's code will reside somewhere in this directory. You may also wish to explore the `app/config` directory and the configuration options that are available to you.

Routing

To get started, let's create our first route. In Laravel, the simplest route is a route to a Closure. Pop open the `app/routes.php` file and add the following route to the bottom of the file:

```
Route::get('users', function()
{
    return 'Users!';
});
```

Now, if you hit the `/users` route in your web browser, you should see `Users!` displayed as the response. Great! You've just created your first route.

Routes can also be attached to controller classes. For example:

```
Route::get('users', 'UserController@getIndex');
```

This route informs the framework that requests to the `/users` route should call the `getIndex` method on the `UserController` class. For more information on controller routing, check out the controller documentation.

Creating A View

Next, we'll create a simple view to display our user data. Views live in the `app/views` directory and contain the HTML of your application. We're going to place two new views in this directory: `layout.blade.php` and `users.blade.php`. First, let's create our `layout.blade.php` file:

```
<html>
    <body>
        <h1>Laravel Quickstart</h1>

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

Next, we'll create our `users.blade.php` view:

```
@extends('layout')

@section('content')
    Users!
@stop
```

Some of this syntax probably looks quite strange to you. That's because we're using Laravel's templating system: Blade. Blade is very fast, because it is simply a handful of regular expressions that are run against your templates to compile them to pure PHP. Blade provides powerful functionality like template inheritance, as well as some syntax sugar on typical PHP control structures such as `if` and `for`. Check out the Blade documentation for more details.

Now that we have our views, let's return it from our `/users` route. Instead of returning `Users!` from the route, return the view instead:

```
Route::get('users', function()
{
    return View::make('users');
});
```

Wonderful! Now you have setup a simple view that extends a layout. Next, let's start working on our database layer.

Creating A Migration

To create a table to hold our data, we'll use the Laravel migration system. Migrations let you expressively define modifications to your database, and easily share them with the rest of your team.

First, let's configure a database connection. You may configure all of your database connections from the `app/config/database.php` file. By default, Laravel is configured to use MySQL, and you will need to supply connection credentials within the database configuration file. If you wish, you may change the `driver` option to `sqlite` and it will use the SQLite database included in the `app/database` directory.

Next, to create the migration, we'll use the Artisan CLI. From the root of your project, run the following from your terminal:

```
php artisan migrate:make create_users_table
```

Next, find the generated migration file in the `app/database/migrations` folder. This file contains a class with two methods: `up` and `down`. In the `up` method, you should make the desired changes to your database tables, and in the `down` method you simply reverse them.

Let's define a migration that looks like this:

```
public function up()
{
```

```
    Schema::create('users', function($table)
    {
        $table->increments('id');
        $table->string('email')->unique();
        $table->string('name');
        $table->timestamps();
    });
}

public function down()
{
    Schema::drop('users');
}
```

Next, we can run our migrations from our terminal using the `migrate` command. Simply execute this command from the root of your project:

```
php artisan migrate
```

If you wish to rollback a migration, you may issue the `migrate:rollback` command. Now that we have a database table, let's start pulling some data!

Eloquent ORM

Laravel ships with a superb ORM: Eloquent. If you have used the Ruby on Rails framework, you will find Eloquent familiar, as it follows the ActiveRecord ORM style of database interaction.

First, let's define a model. An Eloquent model can be used to query an associated database table, as well as represent a given row within that table. Don't worry, it will all make sense soon! Models are typically stored in the `app/models` directory. Let's define a `User.php` model in that directory like so:

```
class User extends Eloquent {}
```

Note that we do not have to tell Eloquent which table to use. Eloquent has a variety of conventions, one of which is to use the plural form of the model name as the model's database table. Convenient!

Using your preferred database administration tool, insert a few rows into your `users` table, and we'll use Eloquent to retrieve them and pass them to our view.

Now let's modify our `/users` route to look like this:

```
Route::get('users', function()
{
    $users = User::all();

    return View::make('users')->with('users', $users);
});
```

Let's walk through this route. First, the `all` method on the `User` model will retrieve all of the rows in the `users` table. Next, we're passing these records to the view via the `with` method. The `with` method accepts a key and a value, and is used to make a piece of data available to a view.

Awesome. Now we're ready to display the users in our view!

Displaying Data

Now that we have made the `users` available to our view. We can display them like so:

```
@extends('layout')

@section('content')
    @foreach($users as $user)
        <p>{{ $user->name }}</p>
    @endforeach
@stop
```

You may be wondering where to find our `echo` statements. When using Blade, you may echo data by surrounding it with double curly braces. It's a cinch. Now, you should be able to hit the `/users` route and see the names of your users displayed in the response.

This is just the beginning. In this tutorial, you've seen the very basics of Laravel, but there are so many more exciting things to learn. Keep reading through the documentation and dig deeper into the powerful features available to you in Eloquent and Blade. Or, maybe you're more interested in Queues and Unit Testing. Then again, maybe you want to flex your architecture muscles with the IoC Container. The choice is yours!

Contributing To Laravel

- Introduction
- Pull Requests
- Coding Guidelines

Introduction

Laravel is free, open-source software, meaning anyone can contribute to its development and progress. Laravel source code is currently hosted on Github, which provides an easy method for forking the project and merging your contributions.

Pull Requests

The pull request process differs for new features and bugs. Before sending a pull request for a new feature, you should first create an issue with `[Proposal]` in the title. The proposal should describe the new feature, as well as implementation ideas. The proposal will then be reviewed and either approved or denied. Once a proposal is approved, a pull request may be created implementing the new feature. Pull requests which do not follow this guideline will be closed immediately.

Pull requests for bugs may be sent without creating any proposal issue. If you believe that you know of a solution for a bug that has been filed on Github, please leave a comment detailing your proposed fix.

Additions and corrections to the documentation may also be contributed via the documentation repository on Github.

Feature Requests

If you have an idea for a new feature you would like to see added to Laravel, you may create an issue on Github with `[Request]` in the title. The feature request will then be reviewed by a core contributor.

Coding Guidelines

Laravel follows the PSR-0 and PSR-1 coding standards. In addition to these standards, below is a list of other coding standards that should be followed:

- Namespace declarations should be on the same line as `<?php`.
- Class opening `{` should be on the same line as the class name.
- Function and control structure opening `{` should be on a separate line.
- Interface names are suffixed with `Interface` (`FooInterface`)

Installation

- Install Composer
- Install Laravel

- Server Requirements
- Configuration
- Pretty URLs

Install Composer

Laravel utilizes Composer to manage its dependencies. First, download a copy of the `composer.phar`. Once you have the PHAR archive, you can either keep it in your local project directory or move to `usr/local/bin` to use it globally on your system. On Windows, you can use the Composer Windows installer.

Install Laravel

Via Composer Create-Project

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

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

Via Download

Once Composer is installed, download the latest version of the Laravel framework and extract its contents into a directory on your server. Next, in the root of your Laravel application, run the `php composer.phar install` (or `composer install`) command to install all of the framework's dependencies. This process requires Git to be installed on the server to successfully complete the installation.

If you want to update the Laravel framework, you may issue the `php composer.phar update` command.

Server Requirements

The Laravel framework has a few system requirements:

- PHP >= 5.3.7
- MCrypt PHP Extension

Configuration

Laravel needs almost no configuration out of the box. You are free to get started developing! However, you may wish to review the `app/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.

Permissions

Laravel requires one set of permissions to be configured - folders within app/storage require write access by the web server.

Paths

Several of the framework directory paths are configurable. To change the location of these directories, check out the `bootstrap/paths.php` file.

> **Note:** Laravel is designed to protect your application code, and local storage by placing only files that are necessarily public in the public folder. It is recommended that you either set the public folder as your site's documentRoot (also known as a web root) or to place the contents of public into your site's root directory and place all of Laravel's other files outside the web root.

Pretty URLs

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]
```

Configuration

Introduction

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

Sometimes you may need to access configuration values at run-time. You may do so using the `Config` class:

Accessing A Configuration Value

```
Config::get('app.timezone');
```

You may also specify a default value to return if the configuration option does not exist:

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

Notice that "dot" style syntax may be used to access values in the various files. You may also set configuration values at run-time:

Setting A Configuration Value

```
Config::set('database.default', 'sqlite');
```

Configuration values that are set at run-time are only set for the current request, and will not be carried over to subsequent requests.

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 on your local development machine than on the production server. It is easy to accomplish this using environment based configuration.

Simply create a folder within the `config` directory that matches your environment name, such as `local`. Next, create the configuration files you wish to override and specify the options for that environment. For example, to override the cache driver for the local environment, you would create a `cache.php` file in `app/config/local` with the following content:

```
<?php

return array(

    'driver' => 'file',

);
```

> **Note:** Do not use 'testing' as an environment name. This is reserved for unit testing.

Notice that you do not have to specify *every* option that is in the base configuration file, but only the options you wish to override. The environment configuration files will "cascade" over the base files.

Next, we need to instruct the framework how to determine which environment it is running in. The default environment is always `production`. However, you may setup other environments within the `bootstrap/start.php` file at the root of your installation. In this file you will find an `$app->detectEnvironment` call. The array passed to this method is used to determine the current environment. You may add other environments and machine names to the array as needed.

```
<?php

$env = $app->detectEnvironment(array(

    'local' => array('your-machine-name'),

));
```

You may also pass a `Closure` to the `detectEnvironment` method, allowing you to implement your own environment detection:

```
$env = $app->detectEnvironment(function()
{
    return $_SERVER['MY_LARAVEL_ENV'];
});
```

You may access the current application environment via the `environment` method:

Accessing The Current Application Environment

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

Maintenance Mode

When your application is in maintenance mode, a custom view will be displayed for all routes into your application. This makes it easy to "disable" your application while it is updating. A call to the `App::down` method is already present in your `app/start/global.php` file. The response from this method will be sent to users when your application is in maintenance mode.

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

```
php artisan down
```

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

```
php artisan up
```

To show a custom view when your application is in maintenance mode, you may add something like the following to your application's `app/start/global.php` file:

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

Request Lifecycle

Overview

The Laravel request lifecycle is fairly simple. A request enters your application and is dispatched to the appropriate route or controller. The response from that route is then sent back to the browser and displayed on the screen. Sometimes you may wish to do some processing before or after your routes are actually called. There are several opportunities to do this, two of which are "start" files and application events.

Start Files

Your application's start files are stored at `app/start`. By default, three are included with your application: `global.php`, `local.php`, and `artisan.php`. For more information about `artisan.php`, refer to the documentation on the Artisan command line.

The `global.php` start file contains a few basic items by default, such as the registration of the Logger and the inclusion of your `app/filters.php` file. However, you are free to add anything to this file that you wish. It will be automatically included on *every* request to your application, regardless of environment. The `local.php` file, on the other hand, is only called when the application is executing in the `local` environment. For more information on environments, check out the configuration documentation.

Of course, if you have other environments in addition to `local`, you may create start files for those environments as well. They will be automatically included when your application is running in that environment.

Application Events

You may also do pre and post request processing by registering `before`, `after`, `close`, `finish`, and `shutdown` application events:

Registering Application Events

```
App::before(function()
{
    //
});

App::after(function($request, $response)
{
    //
});
```

Listeners to these events will be run `before` and `after` each request to your application.

Routing

- Basic Routing
- Route Parameters
- Route Filters

- Named Routes
- Route Groups
- Sub-Domain Routing
- Route Prefixing
- Route Model Binding
- Throwing 404 Errors
- Routing To Controllers

Basic Routing

Most of the routes for your application will be defined in the `app/routes.php` file. The simplest Laravel routes consist of a URI and a Closure callback.

Basic GET Route

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

Basic POST Route

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

Registering A Route Responding To Any HTTP Verb

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

Forcing A Route To Be Served Over HTTPS

```
Route::get('foo', array('https', function()
{
    return 'Must be over HTTPS';
}));
```

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

```
$url = URL::to('foo');
```

Route Parameters

```
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 Defaults

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

Regular Expression Route Constraints

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

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

Of course, you may pass an array of constraints when necessary:

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

Route Filters

Route filters provide a convenient way of limiting access to a given route, which is useful for creating areas of your site which require authentication. There are several filters included in the Laravel framework, including an `auth` filter, an `auth.basic` filter, a `guest` filter, and a `csrf`filter. These are located in the `app/filters.php` file.

Defining A Route Filter

```
Route::filter('old', function()
{
    if (Input::get('age') < 200)
    {
        return Redirect::to('home');
    }
});
```

If a response is returned from a filter, that response will be considered the response to the request and the route will not be executed, and any `after` filters on the route will also be cancelled.

Attaching A Filter To A Route

```
Route::get('user', array('before' => 'old', function()
{
    return 'You are over 200 years old!';
}));
```

Attaching Multiple Filters To A Route

```
Route::get('user', array('before' => 'auth|old', function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

Specifying Filter Parameters

```
Route::filter('age', function($route, $request, $value)
{
    //
});

Route::get('user', array('before' => 'age:200', function()
{
    return 'Hello World';
}));
```

After filters receive a `$response` as the third argument passed to the filter:

```
Route::filter('log', function($route, $request, $response, $value)
{
    //
});
```

Pattern Based Filters

You may also specify that a filter applies to an entire set of routes based on their URI.

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

Route::when('admin/*', 'admin');
```

In the example above, the `admin` filter would be applied to all routes beginning with `admin/`. The asterisk is used as a wildcard, and will match any combination of characters.

You may also constrain pattern filters by HTTP verbs:

```
Route::when('admin/*', 'admin', array('post'));
```

Filter Classes

For advanced filtering, you may wish to use a class instead of a Closure. Since filter classes are resolved out of the application IoC Container, you will be able to utilize dependency injection in these filters for greater testability.

Defining A Filter Class

```
class FooFilter {

    public function filter()
    {
        // Filter logic...
    }

}
```

Registering A Class Based Filter

```
Route::filter('foo', 'FooFilter');
```

Named Routes

Named routes make referring to routes when generating redirects or URLs more convenient. You may specify a name for a route like so:

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

You may also specify route names for controller actions:

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

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

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

$redirect = Redirect::route('profile');
```

You may access the name of a route that is running via the `currentRouteName` method:

```
$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(array('before' => 'auth'), function()
{
    Route::get('/', function()
    {
        // Has Auth Filter
    });

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

Sub-Domain Routing

Laravel routes are also able to handle wildcard sub-domains, and pass you wildcard parameters from the domain:

Registering Sub-Domain Routes

```
Route::group(array('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:

Prefixing Grouped Routes

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

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

});
```

Route Model Binding

Model binding provides a convenient way to inject model instances into your routes. For example, instead of injecting a user's ID, you can inject the entire User model instance that matches the given ID. First, use the `Route::model` method to specify the model that should be used for a given parameter:

Binding A Parameter To A Model

```
Route::model('user', 'User');
```

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

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

Since we have bound the `{user}` parameter to the `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, you may pass a Closure as the third argument to the `model` method:

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

Sometimes you may wish to use your own resolver for route parameters. Simply use the `Route::bind` method:

```
Route::bind('user', function($value, $route)
{
    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 `App::abort` method:

```
App::abort(404);
```

Second, you may 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.

Routing To Controllers

Laravel allows you to not only route to Closures, but also to controller classes, and even allows the creation of resource controllers.

See the documentation on Controllers for more details.

Requests & Input

- Basic Input
- Cookies
- Old Input
- Files
- Request Information

Basic Input

You may access all user input with a few simple methods. 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.

Retrieving An Input Value

```
$name = Input::get('name');
```

Retrieving A Default Value If The Input Value Is Absent

```
$name = Input::get('name', 'Sally');
```

Determining If An Input Value Is Present

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

Getting All Input For The Request

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

Getting Only Some Of The Request Input

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

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

Some JavaScript libraries such as Backbone may send input to the application as JSON. You may access this data via `Input::get` like normal.

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 = Cookie::get('name');
```

Attaching A New Cookie To A Response

```
$response = Response::make('Hello World');

$response->withCookie(Cookie::make('name', 'value', $minutes));
```

Creating A Cookie That Lasts Forever

```
$cookie = Cookie::forever('name', 'value');
```

Old Input

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

Flashing Input To The Session

```
Input::flash();
```

Flashing Only Some Input To The Session

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

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

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::to('form')->withInput();

return Redirect::to('form')->withInput(Input::except('password'));
```

> **Note:** You may flash other data across requests using the Session class.

Retrieving Old Data

```
Input::old('username');
```

Files

Retrieving An Uploaded File

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

Determining If A File Was Uploaded

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

The object returned by the `file` method is an instance of the `Symfony\Component\HttpFoundation\File\Uploa` class, which extends the PHP `SplFileInfo` class and provides a variety of methods for interacting with the file.

Moving An Uploaded File

```
Input::file('photo')->move($destinationPath);

Input::file('photo')->move($destinationPath, $fileName);
```

Retrieving The Path To An Uploaded File

```
$path = Input::file('photo')->getRealPath();
```

Retrieving The Original Name Of An Uploaded File

```
$name = Input::file('photo')->getClientOriginalName();
```

Retrieving The Extension Of An Uploaded File

```
$extension = Input::file('photo')->getClientOriginalExtension();
```

Retrieving The Size Of An Uploaded File

```
$size = Input::file('photo')->getSize();
```

Retrieving The MIME Type Of An Uploaded File

```
$mime = Input::file('photo')->getMimeType();
```

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();
```

Determining If The Request Path Matches A Pattern

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

Get The Request URL

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

Retrieve A Request URI Segment

```
$segment = Request::segment(1);
```

Retrieving A Request Header

```
$value = Request::header('Content-Type');
```

Retrieving Values From $_SERVER

```
$value = Request::server('PATH_INFO');
```

Determine If The Request Is Using AJAX

```
if (Request::ajax())
{
    //
}
```

Determining If The Request Is Over HTTPS

```
if (Request::secure())
{
    //
}
```

Views & Responses

Basic Responses

Returning Strings From Routes

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

Creating Custom Responses

A `Response` instance inherits from the `Symfony\Component\HttpFoundation\Response` class, providing a variety of methods for building HTTP responses.

```
$response = Response::make($contents, $statusCode);

$response->header('Content-Type', $value);

return $response;
```

Attaching Cookies To Responses

```
$cookie = Cookie::make('name', 'value');

return Response::make($content)->withCookie($cookie);
```

Redirects

Returning A Redirect

```
return Redirect::to('user/login');
```

Returning A Redirect With Flash Data

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

> **Note:** Since the `with` method flashes data to the session, you may retrieve the data using the typical `Session::get` method.

Returning A Redirect To A Named Route

```
return Redirect::route('login');
```

Returning A Redirect To A Named Route With Parameters

```
return Redirect::route('profile', array(1));
```

Returning A Redirect To A Named Route Using Named Parameters

```
return Redirect::route('profile', array('user' => 1));
```

Returning A Redirect To A Controller Action

```
return Redirect::action('HomeController@index');
```

Returning A Redirect To A Controller Action With Parameters

```
return Redirect::action('UserController@profile', array(1));
```

Returning A Redirect To A Controller Action Using Named Parameters

```
return Redirect::action('UserController@profile', array('user' => 1));
```

Views

Views typically contain the HTML of your application and provide a convenient way of separating your controller and domain logic from your presentation logic. Views are stored in the `app/views` directory.

A simple view could look something like this:

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

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

This view may be returned to the browser like so:

```
Route::get('/', function()
{
    return View::make('greeting', array('name' => 'Taylor'));
});
```

The second argument passed to `View::make` is an array of data that should be made available to the view.

Passing Data To Views

```
$view = View::make('greeting')->with('name', 'Steve');
```

In the example above the variable `$name` would be accessible from the view, and would contain `Steve`.

If you wish, you may pass an array of data as the second parameter given to the `make` method:

```
$view = View::make('greetings', $data);
```

You may also share a piece of data across all views:

```
View::share('name', 'Steve');
```

Passing A Sub-View To A View

Sometimes you may wish to pass a view into another view. For example, given a sub-view stored at `app/views/child/view.php`, we could pass it to another view like so:

```
$view = View::make('greeting')->nest('child', 'child.view');

$view = View::make('greeting')->nest('child', 'child.view', $data);
```

The sub-view can then be rendered from the parent view:

```
<html>
    <body>
        <h1>Hello!</h1>
        <?php echo $child; ?>
    </body>
</html>
```

View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want bound to a given view each time that view is rendered throughout your application, a view composer can organize that code into a single location. Therefore, view composers may function like "view models" or "presenters".

Defining A View Composer

```
View::composer('profile', function($view)
{
    $view->with('count', User::count());
});
```

Now each time the `profile` view is rendered, the `count` data will be bound to the view.

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

```
View::composer(array('profile','dashboard'), function($view)
{
    $view->with('count', User::count());
});
```

If you would rather use a class based composer, which will provide the benefits of being resolved through the application IoC Container, you may do so:

```
View::composer('profile', 'ProfileComposer');
```

A view composer class should be defined like so:

```
class ProfileComposer {

    public function compose($view)
    {
        $view->with('count', User::count());
    }

}
```

Note that there is no convention on where composer classes may be stored. You are free to store them anywhere as long as they can be autoloaded using the directives in your `composer.json` file.

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, simple use the `creator` method:

```
View::creator('profile', function($view)
{
    $view->with('count', User::count());
});
```

Special Responses

Creating A JSON Response

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'));
```

Creating A JSONP Response

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'))->setCallback(Input::get('ca
```

Creating A File Download Response

```
return Response::download($pathToFile);
```

```
return Response::download($pathToFile, $name, $headers);
```

Controllers

Basic Controllers

Instead of defining all of your route-level logic in a single `routes.php` file, you may wish to organize this behavior using Controller classes. Controllers can group related route logic into a class, as well as take advantage of more advanced framework features such as automatic dependency injection.

Controllers are typically stored in the `app/controllers` directory, and this directory is registered in the `classmap` option of your `composer.json` file by default.

Here is an example of a basic controller class:

```
class UserController extends BaseController {

    /**
     * Show the profile for the given user.
     */
    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

All controllers should extend the `BaseController` class. The `BaseController` is also stored in the `app/controllers` directory, and may be used as a place to put shared controller logic. The `BaseController` extends the framework's `Controller` class. Now, We can route to this controller action like so:

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

If you choose to nest or organize your controller using PHP namespaces, simply use the fully qualified class name when defining the route:

```
Route::get('foo', 'Namespace\FooController@method');
```

You may also specify names on controller routes:

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

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

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

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

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

Controller Filters

Filters may be specified on controller routes similar to "regular" routes:

```
Route::get('profile', array('before' => 'auth',
            'uses' => 'UserController@showProfile'));
```

However, you may also specify filters from within your controller:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
```

```
        $this->beforeFilter('auth');

        $this->beforeFilter('csrf', array('on' => 'post'));

        $this->afterFilter('log', array('only' =>
                            array('fooAction', 'barAction')));
    }

}
```

You may also specify controller filters inline using a Closure:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter(function()
        {
            //
        });
    }

}
```

RESTful Controllers

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

Defining A RESTful Controller

```
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()
    {
        //
    }

}
```

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() {}
```

Resource Controllers

Resource controllers make it easier to build RESTful controllers around resources. For example, you may wish to create a controller that manages "photos" stored by your application. Using the `controller:make` command via the Artisan CLI and the `Route::resource` method, we can quickly create such a controller.

To create the controller via the command line, execute the following command:

```
php artisan controller:make PhotoController
```

Now we can 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 stubbed methods for each of these actions with 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/{id}	show	resource.show
GET	/resource/{id}/edit	edit	resource.edit
PUT/PATCH	/resource/{id}	update	resource.update
DELETE	/resource/{id}	destroy	resource.destroy

Sometimes you may only need to handle a subset of the resource actions:

```
php artisan controller:make PhotoController --only=index,show

php artisan controller:make PhotoController --except=index
```

And, you may also specify a subset of actions to handle on the route:

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

Route::resource('photo', 'PhotoController',
                array('except' => array('create', 'store', 'update', 'delete')));
```

Handling Missing Methods

A catch-all method may be defined which will be called when no other matching method is found on a given controller. The method should be named `missingMethod`, and receives the parameter array for the request as its only argument:

Defining A Catch-All Method

```
public function missingMethod($parameters)
{
    //
}
```

Errors & Logging

Error Detail

By default, error detail is enabled for your application. This means that when an error occurs you will be shown an error page with a detailed stack trace and error message. You may turn off error details by setting the `debug` option in your `app/config/app.php` file to `false`. **It is strongly recommended that you turn off error detail in a production environment.**

Handling Errors

By default, the `app/start/global.php` file contains an error handler for all exceptions:

```
App::error(function(Exception $exception)
{
    Log::error($exception);
});
```

This is the most basic error handler. However, you may specify more handlers if needed. Handlers are called based on the type-hint of the Exception they handle. For example, you may create a handler that only handles `RuntimeException` instances:

```
App::error(function(RuntimeException $exception)
{
    // Handle the exception...
});
```

If an exception handler returns a response, that response will be sent to the browser and no other error handlers will be called:

```
App::error(function(InvalidUserException $exception)
{
    Log::error($exception);
```

```
    return 'Sorry! Something is wrong with this account!';
});
```

To listen for PHP fatal errors, you may use the `App::fatal` method:

```
App::fatal(function($exception)
{
    //
});
```

HTTP Exceptions

Exceptions in respect to HTTP, refer to errors that may occur during a client request. This may be a page not found error (404), an unauthorized error (401) or even a generated 500 error. In order to return such a response, use the following:

```
App::abort(404, 'Page not found');
```

The first argument, is the HTTP status code, with the following being a custom message you'd like to show with the error.

In order to raise a 401 Unauthorized exception, just do the following:

```
App::abort(401, 'You are not authorized.');
```

These exceptions can be executed at any time during the request's lifecycle.

Handling 404 Errors

You may register an error handler that handles all "404 Not Found" errors in your application, allowing you to return custom 404 error pages:

```
App::missing(function($exception)
{
    return Response::view('errors.missing', array(), 404);
});
```

Logging

The Laravel logging facilities provide a simple layer on top of the powerful Monolog. By default, Laravel is configured to create daily log files for your application, and these files are stored in `app/storage/logs`. You may write information to these logs 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', array('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 Listener

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

Cache

Configuration

Laravel provides a unified API for various caching systems. The cache configuration is located at `app/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.

Cache Usage

Storing An Item In The Cache

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

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

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

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.

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 Sections

> **Note:** Cache sections are not supported when using the `file` or `database` cache drivers.

Cache sections allow you to group related items in the cache, and then flush the entire section. To access a section, use the `section` method:

Accessing A Cache Section

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

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

You may also access cached items from the section, as well as use the other cache methods such as `increment` and `decrement`:

Accessing Items In A Cache Section

```
$anne = Cache::section('people')->get('Anne');
```

Then you may flush all items in the section:

```
Cache::section('people')->flush();
```

Database Cache

When using the `database` cache driver, you will need to setup a table to contain the cache items. Below is an example `Schema` declaration for the table:

```
Schema::create('cache', function($table)
{
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

Events

Basic Usage

The Laravel `Event` class provides a simple observer implementation, allowing you to subscribe and listen for events in your application.

Subscribing To An Event

```
Event::listen('user.login', function($user)
{
    $user->last_login = new DateTime;

    $user->save();
});
```

Firing An Event

```
$event = Event::fire('user.login', array($user));
```

You may also specify a priority when subscribing to events. Listeners with higher priority will be run first, while listeners that have the same priority will be run in order of subscription.

Subscribing To Events With Priority

```
Event::listen('user.login', 'LoginHandler', 10);

Event::listen('user.login', 'OtherHandler', 5);
```

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

Stopping The Propagation Of An Event

```
Event::listen('user.login', function($event)
{
    // Handle the event...

    return false;
});
```

Wildcard Listeners

When registering an event listener, you may use asterisks to specify wildcard listeners:

Registering Wildcard Event Listeners

```
Event::listen('foo.*', function($param, $event)
{
    // Handle the event...
});
```

This listener will handle all events that begin with `foo.`. Note that the full event name is passed as the last argument to the handler.

Using Classes As Listeners

In some cases, you may wish to use a class to handle an event rather than a Closure. Class event listeners will be resolved out of the Laravel IoC container, providing you the full power of dependency injection on your listeners.

Registering A Class Listener

```
Event::listen('user.login', 'LoginHandler');
```

By default, the `handle` method on the `LoginHandler` class will be called:

Defining An Event Listener Class

```
class LoginHandler {

    public function handle($data)
    {
        //
    }

}
```

If you do not wish to use the default `handle` method, you may specify the method that should be subscribed:

Specifying Which Method To Subscribe

```
Event::listen('user.login', 'LoginHandler@onLogin');
```

Queued Events

Using the `queue` and `flush` methods, you may "queue" an event for firing, but not fire it immediately:

Registering A Queued Event

```
Event::queue('foo', array($user));
```

Registering An Event Flusher

```
Event::flusher('foo', function($user)
{
    //
});
```

Finally, you may run the "flusher" and flush all queued events using the `flush` method:

```
Event::flush('foo');
```

Event Subscribers

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:

Defining An Event Subscriber

```
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('user.login', 'UserEventHandler@onUserLogin');

        $events->listen('user.logout', 'UserEventHandler@onUserLogout');
    }

}
```

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

Registering An Event Subscriber

```
$subscriber = new UserEventHandler;

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

Extending The Framework

Introduction

Laravel offers many extension points for you to customize the behavior of the framework's core components, or even replace them entirely. For example, the hashing facilities are defined by a `HasherInterface` contract, which you may implement based on your application's requirements. You may also extend the `Request` object, allowing you to add your own convenient "helper" methods. You may even add entirely new authentication, cache, and session drivers!

Laravel components are generally extended in two ways: binding new implementations in the IoC container, or registering an extension with a `Manager` class, which are implementations of the "Factory" design pattern. In this chapter we'll explore the various methods of extending the framework and examine the necessary code.

> **Note:** Remember, Laravel components are typically extended in one of two ways: IoC bindings and the `Manager` classes. The manager classes serve as an implementation of the "factory" design pattern, and are responsible for instantiating driver based facilities such as cache and session.

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, Native, 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 Illuminate\Cache\Repository instance...
});
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `app/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.

To create our custom cache driver, we first need to implement the `Illuminate\Cache\StoreInterface` contract. So, our MongoDB cache implementation would look something like this:

```
class MongoStore implements Illuminate\Cache\StoreInterface {

    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:

```
use Illuminate\Cache\Repository;

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

As you can see in the example above, you may use the base `Illuminate\Cache\Repository` when creating custom cache drivers. There is typically no need to create your own repository class.

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 application's primary folder. For example, if the application is named `Snappy`, you could place the cache extension in `app/Snappy/Extensions/MongoStore.php`. 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.

> **Note:** If you're ever wondering where to put a piece of code, always consider a service provider. As we've discussed, using a service provider to organize framework extensions is a great way to organize your code.

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
});
```

Note that our custom cache driver should implement the `SessionHandlerInterface`. This interface is included in the PHP 5.4+ core. If you are using PHP 5.3, the interface will be defined for you by Laravel so you have forward-compatibility. 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 `native` session driver that uses PHP's native file storage for sessions, 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 `app/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\Auth\UserProviderInterface
});
```

The `UserProviderInterface` implementations are only responsible for fetching a `UserInterface` 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 `UserProviderInterface`:

```
interface UserProviderInterface {
```

```
    public function retrieveById($identifier);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(UserInterface $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 `UserInterface` implementation matching the ID should be retrieved and returned by the method.

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 `$credentails['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 `UserProviderInterface`, let's take a look at the `UserInterface`. Remember, the provider should return implementations of this interface from the `retrieveById` and `retrieveByCredentials` methods:

```
interface UserInterface {

    public function getAuthIdentifier();
    public function getAuthPassword();

}
```

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/models` directory which implements this interface, so you may consult this class for an implementation example.

Finally, once we have implemented the `UserProviderInterface`, 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 `app/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 `app/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 `PaginationServiceProvider` binds a `paginator` key into the IoC container, which resolves into a `Illuminate\Pagination\Environment` instance. You can easily extend and override this class within your own application by overriding this IoC binding. For example, you could create a class that extend the base `Environment`:

```
namespace Snappy\Extensions\Pagination;

class Environment extends \Illuminate\Pagination\Environment {

    //

}
```

Once you have created your class extension, you may create a new `SnappyPaginationProvider` service provider class which overrides the paginator in its `boot` method:

```
class SnappyPaginationProvider extends PaginationServiceProvider {

    public function boot()
    {
        App::bind('paginator', function()
        {
            return new Snappy\Extensions\Pagination\Environment;
        });

        parent::boot();
```

```
    }

}
```

Note that this class extends the `PaginationServiceProvider`, not the default `ServiceProvider` base class. Once you have extended the service provider, swap out the `PaginationServiceProvider` in your `app/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.

Request Extension

Because it is such a foundational piece of the framework and is instantiated very early in the request cycle, extending the `Request` class works a little differently than the previous examples.

First, extend the class like normal:

```
<?php namespace QuickBill\Extensions;

class Request extends \Illuminate\Http\Request {

    // Custom, helpful methods here...

}
```

Once you have extended the class, open the `bootstrap/start.php` file. This file is one of the very first files to be included on each request to your application. Note that the first action performed is the creation of the Laravel `$app` instance:

```
$app = new \Illuminate\Foundation\Application;
```

When a new application instance is created, it will create a new `Illuminate\Http\Request` instance and bind it to the IoC container using the `request` key. So, we need a way to specify a custom class that should be used as the "default" request type, right? And, thankfully, the `requestClass` method on the application instance does just this! So, we can add this line at the very top of our `bootstrap/start.php` file:

```
use Illuminate\Foundation\Application;

Application::requestClass('QuickBill\Extensions\Request');
```

Once you have specified the custom request class, Laravel will use this class anytime it creates a `Request` instance, conveniently allowing you to always have an instance of your custom request class available, even in unit tests!

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!

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.

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');
```

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:

```
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 `app/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 `app/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 Models 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.

Forms & HTML

Opening A Form

Opening A Form

```
{{ Form::open(array('url' => 'foo/bar')) }}
    //
{{ Form::close() }}
```

By default, a `POST` method will be assumed; however, you are free to specify another method:

```
echo Form::open(array('url' => 'foo/bar', 'method' => 'put'))
```

> **Note:** Since HTML forms only support `POST` and `GET`, `PUT` and `DELETE` methods will be spoofed by automatically adding a `_method` hidden field to your form.

You may also open forms that point to named routes or controller actions:

```
echo Form::open(array('route' => 'route.name'))

echo Form::open(array('action' => 'Controller@method'))
```

You may pass in route parameters as well:

```
echo Form::open(array('route' => array('route.name', $user->id)))

echo Form::open(array('action' => array('Controller@method', $user->id)))
```

If your form is going to accept file uploads, add a `files` option to your array:

```
echo Form::open(array('url' => 'foo/bar', 'files' => true))
```

CSRF Protection

Laravel provides an easy method of protecting your application from cross-site request forgeries. First, a random token is placed in your user's session. Don't sweat it, this is done automatically. The CSRF token will be added to your forms as a hidden field automatically. However, if you wish to generate the HTML for the hidden field, you may use the `token` method:

Adding The CSRF Token To A Form

```
echo Form::token();
```

Attaching The CSRF Filter To A Route

```
Route::post('profile', array('before' => 'csrf', function()
{
    //
}));
```

Form Model Binding

Often, you will want to populate a form based on the contents of a model. To do so, use the `Form::model` method:

Opening A Model Form

```
echo Form::model($user, array('route' => array('user.update', $user->id)))
```

Now, when you generate a form element, like a text input, the model's value matching the field's name will automatically be set as the field value. So, for example, for a text input named `email`, the user model's `email` attribute would be set as the value. However, there's more! If there is an item in the Session flash data matching the input name, that will take precedence over the model's value. So, the priority looks like this:

1. Session Flash Data (Old Input)
2. Explicitly Passed Value
3. Model Attribute Data

This allows you to quickly build forms that not only bind to model values, but easily re-populate if there is a validation error on the server!

> **Note:** When using `Form::model`, be sure to close your form with `Form::close`!

Labels

Generating A Label Element

```
echo Form::label('email', 'E-Mail Address');
```

Specifying Extra HTML Attributes

```
echo Form::label('email', 'E-Mail Address', array('class' => 'awesome'));
```

> **Note:** After creating a label, any form element you create with a name matching the label name will automatically receive an ID matching the label name as well.

Text, Text Area, Password & Hidden Fields

Generating A Text Input

```
echo Form::text('username');
```

Specifying A Default Value

```
echo Form::text('email', 'example@gmail.com');
```

> **Note:** The *hidden* and *textarea* methods have the same signature as the *text* method.

Generating A Password Input

```
echo Form::password('password');
```

Generating Other Inputs

```
echo Form::email($name, $value = null, $attributes = array());
echo Form::file($name, $attributes = array());
```

Checkboxes and Radio Buttons

Generating A Checkbox Or Radio Input

```
echo Form::checkbox('name', 'value');

echo Form::radio('name', 'value');
```

Generating A Checkbox Or Radio Input That Is Checked

```
echo Form::checkbox('name', 'value', true);

echo Form::radio('name', 'value', true);
```

File Input

Generating A File Input

```
echo Form::file('image');
```

Drop-Down Lists

Generating A Drop-Down List

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'));
```

Generating A Drop-Down List With Selected Default

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'), 'S');
```

Generating A Grouped List

```
echo Form::select('animal', array(
    'Cats' => array('leopard' => 'Leopard'),
    'Dogs' => array('spaniel' => 'Spaniel'),
));
```

Buttons

Generating A Submit Button

```
echo Form::submit('Click Me!');
```

> **Note:** Need to create a button element? Try the *button* method. It has the same signature as *submit*.

Custom Macros

It's easy to define your own custom Form class helpers called "macros". Here's how it works. First, simply register the macro with a given name and a Closure:

Registering A Form Macro

```
Form::macro('myField', function()
{
    return '<input type="awesome">';
});
```

Now you can call your macro using its name:

Calling A Custom Form Macro

```
echo Form::myField();
```

Helper Functions

- Arrays
- Paths
- Strings
- URLs
- Miscellaneous

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

var_dump(array_fetch($array, '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_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')));

$array = array_forget($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');
```

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'];
}));
```

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.

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 `app/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
```

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');
```

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

Tranlate 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 =
```

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();
```

IoC Container

Introduction

The Laravel inversion of control container is a powerful tool for managing class dependencies. Dependency injection is a method of removing hard-coded class dependencies. Instead, the dependencies are injected at run-time, allowing for greater flexibility as dependency implementations may be swapped easily.

Understanding the Laravel IoC container is essential to building a powerful, large application, as well as for contributing to the Laravel core itself.

Basic Usage

There are two ways the IoC container can resolve dependencies: via Closure callbacks or automatic resolution. First, we'll explore Closure callbacks. First, a "type" may be bound into the container:

Binding A Type Into The Container

```
App::bind('foo', function($app)
{
    return new FooBar;
});
```

Resolving A Type From The Container

```
$value = App::make('foo');
```

When the `App::make` method is called, the Closure callback is executed and the result is returned.

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:

Binding A "Shared" Type Into The Container

```
App::singleton('foo', function()
{
    return new FooBar;
});
```

You may also bind an existing object instance into the container using the `instance` method:

Binding An Existing Instance Into The Container

```
$foo = new Foo;

App::instance('foo', $foo);
```

Automatic Resolution

The IoC container is powerful enough to resolve classes without any configuration at all in many scenarios. For example:

Resolving A Class

```
class FooBar {

    public function __construct(Baz $baz)
    {
        $this->baz = $baz;
    }

}

$fooBar = App::make('FooBar');
```

Note that even though we did not register the FooBar class in the container, the container will still be able to resolve the class, even injecting the `Baz` dependency automatically!

When a type is not bound in the container, it will use PHP's Reflection facilities to inspect the class and read the constructor's type-hints. Using this information, the container can automatically build an instance of the class.

However, in some cases, a class may depend on an interface implementation, not a "concrete type". When this is the case, the `App::bind` method must be used to inform the container which interface implementation to inject:

Binding An Interface To An Implementation

```
App::bind('UserRepositoryInterface', 'DbUserRepository');
```

Now consider the following controller:

```
class UserController extends BaseController {

    public function __construct(UserRepositoryInterface $users)
    {
        $this->users = $users;
    }

}
```

Since we have bound the `UserRepositoryInterface` to a concrete type, the `DbUserRepository` will automatically be injected into this controller when it is created.

Practical Usage

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

Type-Hinting Controller Dependencies

```
class OrderController extends BaseController {

    public function __construct(OrderRepository $orders)
    {
        $this->orders = $orders;
    }

    public function getIndex()
    {
        $all = $this->orders->all();

        return View::make('orders', compact('all'));
    }

}
```

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

Filters, composers, and event handlers may also be resolved out of the IoC container. When registering them, simply give the name of the class that should be used:

Other Examples Of IoC Usage

```
Route::filter('foo', 'FooFilter');

View::composer('foo', 'FooComposer');

Event::listen('foo', 'FooHandler');
```

Service Providers

Service providers are a great way to group related IoC registrations in a single location. Think of them as a way to bootstrap components in your application. Within a service provider, you might register a custom authentication driver, register your application's repository classes with the IoC container, or even setup a custom Artisan command.

In fact, most of the core Laravel components include service providers. All of the registered service providers for your application are listed in the `providers` array of the `app/config/app.php` configuration file.

To create a service provider, simply extend the `Illuminate\Support\ServiceProvider` class and define a `register` method:

Defining A Service Provider

```
use Illuminate\Support\ServiceProvider;

class FooServiceProvider extends ServiceProvider {

    public function register()
    {
        $this->app->bind('foo', function()
        {
            return new Foo;
        });
    }

}
```

Note that in the `register` method, the application IoC container is available to you via the `$this->app` property. Once you have created a provider and are ready to register it with your application, simply add it to the `providers` array in your `app` configuration file.

You may also register a service provider at run-time using the `App::register` method:

Registering A Service Provider At Run-Time

```
App::register('FooServiceProvider');
```

Container Events

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

Registering A Resolving Listener

```
App::resolving(function($object)
{
    //
});
```

Note that the object that was resolved will be passed to the callback.

Localization

Introduction

The Laravel `Lang` class 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 `app/lang` directory. Within this directory there should be a subdirectory for each language supported by the application.

```
/app
    /lang
        /en
            messages.php
        /es
            messages.php
```

Language files simply return an array of keyed strings. For example:

Example Language File

```
<?php

return array(
    'welcome' => 'Welcome to our application'
);
```

The default language for your application is stored in the `app/config/app.php` configuration file. You may change the active language at any time using the `App::setLocale` method:

Changing The Default Language At Runtime

```
App::setLocale('es');
```

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);
```

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',
```

Mail

Configuration

Laravel provides a clean, simple API over the popular SwiftMailer library. The mail configuration file is `app/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.

Basic Usage

The `Mail::send` method may be used to send an e-mail message:

```
Mail::send('emails.welcome', $data, function($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` that should be passed to the view, and 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));
```

> **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` class.

Queueing Mail

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` class:

Queueing A Mail Message

```
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 `app/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.

Enabling Pretend Mail Mode

```
Mail::pretend();
```

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. In previous versions of Laravel, these types of packages were called "bundles". These packages may have routes, controllers, views, configuration, and migrations specifically intended to enhance a Laravel application. As no special process is needed to develop stand-alone packages, 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.

Creating A Package

The easiest way to create a new package for use with Laravel is the `workbench` Artisan command. First, you will need to set a few options in the `app/config/workbench.php` file. In that file, you will find a `name` and `email` option. These values will be used to generate a `composer.json` file for your new package. Once you have supplied those values, you are ready to build a workbench package!

Issuing The Workbench Artisan Command

```
php artisan workbench vendor/package --resources
```

The vendor name is a way to distinguish your package from other packages of the same name from different authors. For example, if I (Taylor Otwell) were to create a new package named "Zapper", the vendor name could be `Taylor` while the package name would be `Zapper`. By default, the workbench will create framework agnostic packages; however, the `resources` command tells the workbench to generate the package with Laravel specific directories such as `migrations`, `views`, `config`, etc.

Once the `workbench` command has been executed, your package will be available within the `workbench` directory of your Laravel installation. Next, you should register the `ServiceProvider` that was created for your package. You may register the provider by adding it to the `providers` array in the `app/config/app.php` file. This will instruct Laravel to load your package when your application starts. Service providers use a `[Package]ServiceProvider` naming convention. So, using the example above, you would add `Taylor\Zapper\ZapperServiceProvider` to the `providers` array.

Once the provider has been registered, you are ready to start developing your package! However, before diving in, you may wish to review the sections below to get more familiar with the package structure and development workflow.

> **Note:** If your service provider cannot be found, run the `php artisan dump-autoload` command from your application's root directory

Package Structure

When using the `workbench` command, your package will be setup with conventions that allow the package to integrate well with other parts of the Laravel framework:

Basic Package Directory Structure

```
/src
    /Vendor
        /Package
            PackageServiceProvider.php
    /config
    /lang
    /migrations
    /views
/tests
/public
```

Let's explore this structure further. The `src/Vendor/Package` directory is the home of all of your package's classes, including the `ServiceProvider`. The `config`, `lang`, `migrations`, and `views` directories, as you might guess, contain the corresponding resources for your package. Packages may have any of these resources, just like "regular" applications.

Service Providers

Service providers are simply bootstrap classes for packages. By default, they contain two methods: `boot` and `register`. Within these methods you may do anything you like: include a routes file, register bindings in the IoC container, attach to events, or anything else you wish to do.

The `register` method is called immediately when the service provider is registered, while the `boot` command is only called right before a request is routed. So, if actions in your service provider rely on another service provider already being registered, or you are overriding services bound by another provider, you should use the `boot` method.

When creating a package using the `workbench`, the `boot` command will already contain one action:

```
$this->package('vendor/package');
```

This method allows Laravel to know how to properly load the views, configuration, and other resources for your application. In general, there should be no need for you to change this line of code, as it will setup the package using the workbench conventions.

There is not a "default location" for service provider classes. You may put them anywhere you like, perhaps organizing them in a `Providers` namespace within your `app` directory. The file may be placed anywhere, as long as Composer's auto-loading facilities know how to load the class.

Package Conventions

When utilizing resources from a package, such as configuration items or views, a double-colon syntax will generally be used:

Loading A View From A Package

```
return View::make('package::view.name');
```

Retrieving A Package Configuration Item

```
return Config::get('package::group.option');
```

> **Note:** If your package contains migrations, consider prefixing the migration name with your package name to avoid potential class name conflicts with other packages.

Development Workflow

When developing a package, it is useful to be able to develop within the context of an application, allowing you to easily view and experiment with your templates, etc. So, to get started, install a fresh copy of the Laravel framework, then use the `workbench` command to create your package structure.

After the `workbench` command has created your package. You may `git init` from the `workbench/[vendor]/[package]` directory and `git push` your package straight from the workbench! This will allow you to conveniently develop the package in an application context without being bogged down by constant `composer update` commands.

Since your packages are in the `workbench` directory, you may be wondering how Composer knows to autoload your package's files. When the `workbench` directory exists, Laravel will intelligently scan it for packages, loading their Composer autoload files when the application starts!

If you need to regenerate your package's autoload files, you may use the `php artisan dump-autoload` command. This command will regenerate the autoload files for your root project, as well as any workbenches you have created.

Running The Artisan Autoload Command

```
php artisan dump-autoload
```

Package Routing

In prior versions of Laravel, a `handles` clause was used to specify which URIs a package could respond to. However, in Laravel 4, a package may respond to any URI. 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()
{
    $this->package('vendor/package');

    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.

Package Configuration

Some packages may require configuration files. These files should be defined in the same way as typical application configuration files. And, when using the default `$this->package` method of registering resources in your service provider, may be accessed using the usual "double-colon" syntax:

Accessing Package Configuration Files

```
Config::get('package::file.option');
```

However, if your package contains a single configuration file, you may simply name the file `config.php`. When this is done, you may access the options directly, without specifying the file name:

Accessing Single File Package Configuration

```
Config::get('package::option');
```

Sometimes, you may wish to register package resources such as views outside of the typical `$this->package` method. Typically, this would only be done if the resources were not in a conventional location. To register the resources manually, you may use the `addNamespace` method of the `View`, `Lang`, and `Config` classes:

Registering A Resource Namespace Manually

```
View::addNamespace('package', __DIR__.'/path/to/views');
```

Once the namespace has been registered, you may use the namespace name and the "double colon" syntax to access the resources:

```
return View::make('package::view.name');
```

The method signature for `addNamespace` is identical on the `View`, `Lang`, and `Config` classes.

Cascading Configuration Files

When other developers install your package, they may wish to override some of the configuration options. However, if they change the values in your package source code, they will be overwritten the next time Composer updates the package. Instead, the `config:publish` artisan command should be used:

Executing The Config Publish Command

```
php artisan config:publish vendor/package
```

When this command is executed, the configuration files for your application will be copied to `app/config/packages/vendor/package` where they can be safely modified by the developer!

> **Note:** The developer may also create environment specific configuration files for your package by placing them in `app/config/packages/vendor/package/environment`.

Package Migrations

You may easily create and run migrations for any of your packages. To create a migration for a package in the workbench, use the `--bench` option:

Creating Migrations For Workbench Packages

```
php artisan migrate:make create_users_table --bench="vendor/package"
```

Running Migrations For Workbench Packages

```
php artisan migrate --bench="vendor/package"
```

To run migrations for a finished package that was installed via Composer into the `vendor` directory, you may use the `--package` directive:

Running Migrations For An Installed Package

```
php artisan migrate --package="vendor/package"
```

Package Assets

Some packages may have assets such as JavaScript, CSS, and images. However, we are unable to link to assets in the `vendor` or `workbench` directories, so we need a way to move these assets into the `public` directory of our application. The `asset:publish` command will take care of this for you:

Moving Package Assets To Public

```
php artisan asset:publish
```

```
php artisan asset:publish vendor/package
```

If the package is still in the `workbench`, use the `--bench` directive:

```
php artisan asset:publish --bench="vendor/package"
```

This command will move the assets into the `public/packages` directory according to the vendor and package name. So, a package named `userscape/kudos` would have its assets moved to `public/packages/userscape/kudos`. Using this asset publishing convention allows you to safely code asset paths in your package's views.

Publishing Packages

When your package is ready to publish, you should submit the package to the Packagist repository. If the package is specific to Laravel, consider adding a `laravel` tag to your package's `composer.json` file.

Also, it is courteous and helpful to tag your releases so that developers can depend on stable versions when requesting your package in their `composer.json` files. If a stable version is not ready, consider using the `branch-alias` Composer directive.

Once your package has been published, feel free to continue developing it within the application context created by `workbench`. This is a great way to continue to conveniently develop the package even after it has been published.

Some organizations choose to host their own private repository of packages for their own developers. If you are interested in doing this, review the documentation for the Satis project provided by the Composer team.

Pagination

- Usage
- Appending To Pagination Links

Configuration

In other frameworks, pagination can be very painful. Laravel makes it a breeze. There is a single configuration option in the `app/config/view.php` file. The `pagination` option specifies which view should be used to create pagination links. By default, Laravel includes two views.

The `pagination::slider` view will show an intelligent "range" of links based on the current page, while the `pagination::simple` view will simply show "previous" and "next" buttons. **Both views are compatible with Twitter Bootstrap out of the box.**

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);
```

You may also paginate Eloquent models:

Paginating An Eloquent Model

```
$users = 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 `links` method:

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->links(); ?>
```

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:

- `getCurrentPage`
- `getLastPage`
- `getPerPage`
- `getTotal`
- `getFrom`
- `getTo`

Sometimes you may wish to create a pagination instance manually, passing it an array of items. You may do so using the `Paginator::make` method:

Creating A Paginator Manually

```
$paginator = Paginator::make($items, $totalItems, $perPage);
```

Customizing The Paginator URI

You may also customize the URI used by the paginator via the `setBaseUrl` method:

```
$users = User::paginate();

$users->setBaseUrl('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(array('sort' => 'votes'))->links(); ?>
```

This will generate URLs that look something like this:

```
http://example.com/something?page=2&sort=votes
```

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 `app/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 Beanstalkd, IronMQ, Amazon SQS, and synchronous (for local use) driver.

The following dependencies are needed for the listed queue drivers:

- Beanstalkd: `pda/pheanstalk`
- Amazon SQS: `aws/aws-sdk-php`
- IronMQ: `iron-io/iron_mq`

Basic Usage

To push a new job onto the queue, use the `Queue::push` method:

Pushing A Job Onto The Queue

```
Queue::push('SendEmail', array('message' => $message));
```

The first argument given to the `push` method is the name of the class that should be used to process the job. The second argument is an array of data that should be passed to the handler. A job handler should be defined like so:

Defining A Job Handler

```
class SendEmail {

    public function fire($job, $data)
    {
```

```
        //
    }

}
```

Notice the only method that is required is `fire`, which receives a `Job` instance as well as the array of `data` that was pushed onto the queue.

If you want the job to use a method other than `fire`, you may specify the method when you push the job:

Specifying A Custom Handler Method

```
Queue::push('SendEmail@send', array('message' => $message));
```

Once you have processed a job, it must be deleted from the queue, which can be done via the `delete` method on the `Job` instance:

Deleting A Processed Job

```
public function fire($job, $data)
{
    // Process the job...

    $job->delete();
}
```

If you wish to release a job back onto the queue, you may do so via the `release` method:

Releasing A Job Back Onto The Queue

```
public function fire($job, $data)
{
    // Process the job...

    $job->release();
}
```

You may also specify the number of seconds to wait before the job is released:

```
$job->release(5);
```

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:

Checking The Number Of Run Attempts

```
if ($job->attempts() > 3)
{
    //
}
```

You may also access the job identifier:

Accessing The Job ID

```
$job->getJobId();
```

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:** When pushing Closures onto the queue, the `__DIR__` and `__FILE__` constants should not be used.

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 marshaling 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 also set the length of time (in seconds) each job should be allowed to run:

Specifying The Job Timeout Parameter

```
php artisan queue:listen --timeout=60
```

To process only the first job on the queue, you may use the `queue:work` command:

Processing The First Job On The Queue

```
php artisan queue:work
```

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 `app/config/queue.php` configuration file.

Next, you may use the `queue:subscribe` Artisan command to register a URL end-point that will receive newly pushed queue jobs:

Registering A Push Queue Subscriber

```
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.

Security

Configuration

Laravel aims to make implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at `app/config/auth.php`, which contains several well documented options for tweaking the behavior of the authentication facilities.

By default, Laravel includes a `User` model in your `app/models` directory which may be used with the default Eloquent authentication driver. Please remember when building the Schema for this Model to ensure that the password field is a minimum of 60 characters.

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

Storing Passwords

The Laravel `Hash` class provides secure Bcrypt hashing:

Hashing A Password Using Bcrypt

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

Authenticating Users

To log a user into your application, you may use the `Auth::attempt` method.

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{
    return Redirect::intended('dashboard');
}
```

Take note that `email` is not a required option, it is merely used for example. You should use whatever column name corresponds to a "username" in your database. The `Redirect::intended` function will redirect the user to the URL they were trying 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.

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.

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

Determining If A User Is Authenticated

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

If you would like to provide "remember me" functionality in your application, you may pass `true` as the second argument to the `attempt` method, which will keep the user authenticated indefinitely (or until they manually logout):

Authenticating A User And "Remembering" Them

```
if (Auth::attempt(array('email' => $email, 'password' => $password), true))
{
    // The user is being remembered...
}
```

Note: If the `attempt` method returns `true`, the user is considered logged into the application.

You also may add extra conditions to the authenticating query:

Authenticating A User With Conditions

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

Once a user is authenticated, you may access the User model / record:

Accessing The Logged In User

```
$email = Auth::user()->email;
```

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

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

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

Validating User Credentials Without Login

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

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.

Logging A User In For A Single Request

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

Logging A User Out Of The Application

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

Manually Logging In Users

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

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

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

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

Protecting Routes

Route filters may be used to allow only authenticated users to access a given route. Laravel provides the `auth` filter by default, and it is defined in `app/filters.php`.

Protecting A Route

```
Route::get('profile', array('before' => 'auth', function()
{
    // Only authenticated users may enter...
}));
```

CSRF Protection

Laravel provides an easy method of protecting your application from cross-site request forgeries.

Inserting CSRF Token Into Form

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

Validate The Submitted CSRF Token

```
Route::post('register', array('before' => 'csrf', function()
{
    return 'You gave a valid CSRF token!';
}));
```

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` filter to your route:

Protecting A Route With HTTP Basic

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

By default, the `basic` filter will use the `email` column on the user record when authenticating. If you wish to use another column you may pass the column name as the first parameter to the `basic` method:

```
return Auth::basic('username');
```

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 filter that returns the `onceBasic` method:

Setting Up A Stateless HTTP Basic Filter

```
Route::filter('basic.once', function()
{
    return Auth::onceBasic();
});
```

Password Reminders & Reset

Sending Password Reminders

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\Auth\Reminders\RemindableInterface` contract. Of course, the `User` model included with the framework already implements this interface.

Implementing The RemindableInterface

```
class User extends Eloquent implements RemindableInterface {

    public function getReminderEmail()
    {
        return $this->email;
    }

}
```

Next, a table must be created to store the password reset tokens. To generate a migration for this table, simply execute the `auth:reminders` Artisan command:

Generating The Reminder Table Migration

```
php artisan auth:reminders

php artisan migrate
```

To send a password reminder, we can use the `Password::remind` method:

Sending A Password Reminder

```
Route::post('password/remind', function()
{
    $credentials = array('email' => Input::get('email'));

    return Password::remind($credentials);
});
```

Note that the arguments passed to the `remind` method are similar to the `Auth::attempt` method. This method will retrieve the `User` and send them a password reset link via e-mail. The e-mail view will be passed a `token` variable which may be used to construct the link to the password reset form. The `user` object will also be passed to the view.

> **Note:** You may specify which view is used as the e-mail message by changing the `auth.reminder.email` configuration option. Of course, a default view is provided out of the box.

You may modify the message instance that is sent to the user by passing a Closure as the second argument to the `remind` method:

```
return Password::remind($credentials, function($message, $user)
{
    $message->subject('Your Password Reminder');
});
```

You may also have noticed that we are returning the results of the `remind` method directly from a route. By default, the `remind` method will return a `Redirect` to the current URI. If an error occurred while attempting to reset the password, an `error` variable will be flashed to the session, as well as a `reason`, which can be used to extract a language line from the `reminders` language file. If the password reset was successful, a `success` variable will be flashed to the session. So, your password reset form view could look something like this:

```
@if (Session::has('error'))
    {{ trans(Session::get('reason')) }}
@elseif (Session::has('success'))
    An e-mail with the password reset has been sent.
@endif

<input type="text" name="email">
<input type="submit" value="Send Reminder">
```

Resetting Passwords

Once a user has clicked on the reset link from the reminder e-mail, they should be directed to a form that includes a hidden `token` field, as well as a `password` and `password_confirmation` field. Below is an example route for the password reset form:

```
Route::get('password/reset/{token}', function($token)
{
    return View::make('auth.reset')->with('token', $token);
});
```

And, a password reset form might look like this:

```
@if (Session::has('error'))
    {{ trans(Session::get('reason')) }}
@endif

<input type="hidden" name="token" value="{{ $token }}">
<input type="text" name="email">
<input type="password" name="password">
<input type="password" name="password_confirmation">
```

Again, notice we are using the `Session` to display any errors that may be detected by the framework while resetting passwords. Next, we can define a `POST` route to handle the reset:

```
Route::post('password/reset/{token}', function()
{
    $credentials = array('email' => Input::get('email'));

    return Password::reset($credentials, function($user, $password)
    {
        $user->password = Hash::make($password);

        $user->save();

        return Redirect::to('home');
    });
});
```

If the password reset is successful, the `User` instance and the password will be passed to your Closure, allowing you to actually perform the save operation. Then, you may return a `Redirect` or any other type of response from the Closure which will be returned by the `reset` method. Note that the `reset` method automatically checks for a valid `token` in the request, valid credentials, and matching passwords.

Also, similarly to the `remind` method, if an error occurs while resetting the password, the `reset` method will return a `Redirect` to the current URI with an `error` and `reason`.

Encryption

Laravel provides facilities for strong AES-256 encryption via the mcrypt PHP extension:

Encrypting A Value

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

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

Decrypting A Value

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

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

Setting The Cipher & Mode

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

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

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 `app/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 `native` session driver, which will work well for the majority of applications.

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 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:

- `native` - sessions will be handled by internal PHP session facilities.
- `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

- Controller Layouts
- Blade Templating
- Other Blade Control Structures

Controller Layouts

One method of using templates in Laravel is via controller layouts. By specifying the `layout` property on the controller, the view specified will be created for you and will be the assumed response that should be returned from actions.

Defining A Layout On A Controller

```
class UserController extends BaseController {

    /**
     * The layout that should be used for responses.
     */
    protected $layout = 'layouts.master';

    /**
     * Show the user profile.
     */
    public function showProfile()
    {
        $this->layout->content = View::make('user.profile');
    }

}
```

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 app/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() }}.
```

Of course, all user supplied data should be escaped or purified. To escape the output, you may use the triple curly brace syntax:

```
Hello, {{{ $name }}}.
```

> **Note:** Be very careful when echoing content that is supplied by users of your application. Always use the triple 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

@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', array('some'=>'data'))
```

Overwriting Sections

By default, sections are appended to any previous content that exists in the section. 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 --}}
```

Unit 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. In addition to PHPUnit, Laravel also utilizes the Symfony HttpKernel, DomCrawler, and BrowserKit components to allow you to inspect and manipulate your views while testing, allowing to simulate a web browser.

An example test file is provided in the `app/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 `app/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.

Calling Routes From Tests

You may easily call one of your routes for a test using the `call` method:

Calling A Route From A Test

```
$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());
```

You may also call a controller from a test:

Calling A Controller From A Test

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

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');
```

> **Note:** Route filters are disabled when in the testing environment. To enable them, add `Route::enableFilters()` to your test.

DOM Crawler

You may also call a route and receive a DOM Crawler instance that you may use to inspect the content:

```
$crawler = $this->client->request('GET', '/');

$this->assertTrue($this->client->getResponse()->isOk());

$this->assertCount(1, $crawler->filter('h1:contains("Hello World!")'));
```

For more information on how to use the crawler, refer to its official documentation.

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', array('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(array('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);
}
```

Helper Methods

The `TestCase` class contains several helper methods to make testing your application easier.

You may set the currently authenticated user using the `be` method:

Setting The Currently Authenticated User

```
$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.

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.

Multiple rules may be delimited using either a "pipe" character, or as separate elements of an array.

Using Arrays To Specify Rules

```
$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'
    )
);
```

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.

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 ($validator->fails())
    {
        return Redirect::to('register')->withErrors($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'); ?>
```

Available Validation Rules

Below is a list of all available validation rules and their function:

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Before (Date)
- Between
- Confirmed
- Date
- Date Format
- Different
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer
- IP Address
- Max
- MIME Types
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required Without
- Same

- Size
- Unique (Database)
- URL

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.

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.

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.

email The field under validation must be formatted as an e-mail address.

exists:*table*,*column* The field under validation must exists 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'
```

image The file under validation must be an image (jpeg, png, bmp, or gif)

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 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 *value*.

required_with:*foo,bar,...* The field under validation must be present *only if* the other specified fields are present.

required_without:*foo,bar,...* The field under validation must be present *only when* 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.

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.

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',
);
```

Sometimes you may wish to specify a custom error messages only for a specific field:

Specifying A Custom Message For A Given Attribute

```
$messages = array(
    'email.required' => 'We need to know your e-mail address!',
);
```

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 `app/lang/xx/validation.php` language file.

Specifying Custom Messages In Language Files

```
'custom' => array(
    'email' => array(
        'required' => 'We need to know your e-mail address!',
    ),
),
```

Custom Validation Rules

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using the `Validator::extend` method:

Registering A Custom Validation Rule

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

> **Note:** The name of the rule passed to the `extend` method must be "snake cased".

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.

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`:

Extending The Validator Class

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'foo';
    }

}
```

Next, you need to register your custom Validator extension:

Registering A Custom Validator Resolver

```
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);
}
```

Basic Database Usage

- Running Queries
- Database Transactions
- Accessing Connections
- Query Logging

Configuration

Laravel makes connecting with databases and running queries extremely simple. The database configuration file is `app/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.

Running Queries

Once you have configured your database connection, you may run queries using the `DB` class.

Running A Select Query

```
$results = DB::select('select * from users where id = ?', array(1));
```

The `select` method will always return an `array` of results.

Running An Insert Statement

```
DB::insert('insert into users (id, name) values (?, ?)', array(1, 'Dayle'));
```

Running An Update Statement

```
DB::update('update users set votes = 100 where name = ?', array('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');
```

You may listen for query events using the `DB::listen` method:

Listening For Query Events

```
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(array('votes' => 1));

    DB::table('posts')->delete();
});
```

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');
```

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();
```

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);
}
```

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 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');
```

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();
```

Advanced Wheres

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:

Parameter Grouping

```
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();
```

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'));
```

Inserts

Inserting Records Into A Table

```
DB::table('users')->insert(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

If the table has an auto-incrementing id, use `insertGetId` to insert a record and retrieve the id:

Inserting Records Into A Table With An Auto-Incrementing 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));
```

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:

Performing A Query Union

```
$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`.

Caching Queries

You may easily cache the results of a query using the `remember` method:

Caching A Query Result

```
$users = DB::table('users')->remember(10)->get();
```

In this example, the results of the query will be cached for ten minutes. While the results are cached, the query will not be run against the database, and the results will be loaded from the default cache driver specified for your application.

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 `app/config/database.php`.

Basic Usage

To get started, create an Eloquent model. Models typically live in the `app/models` directory, but you are free to place them anywhere that can be auto-loaded according to your `composer.json` file.

Defining An Eloquent Model

```
class User extends Eloquent {}
```

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 Eloquent {

    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);
}
```

Of course, you may also use the query builder aggregate functions.

Eloquent Aggregates

```
$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();
```

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);
```

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.

The `fillable` property specifies which attributes should be mass-assignable. This can be set at the class or instance level.

Defining Fillable Attributes On A Model

```
class User extends Eloquent {

    protected $fillable = array('first_name', 'last_name', 'email');

}
```

In this example, only the three listed attributes will be mass-assignable.

The inverse of `fillable` is `guarded`, and serves as a "black-list" instead of a "white-list":

Defining Guarded Attributes On A Model

```
class User extends Eloquent {

    protected $guarded = array('id', 'password');

}
```

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 method:

Blocking All Attributes From Mass Assignment

```
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.

Setting The Guarded Attributes On The Model

```
class User extends Eloquent {

    protected $guarded = array('id', 'account_id');

}
```

Using The Model Create Method

```
$user = User::create(array('name' => 'John'));
```

To update a model, you may retrieve it, change an attribute, and use the `save` method:

Updating A Retrieved Model

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

$user->email = 'john@foo.com';

$user->save();
```

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:

Saving A Model And Relationships

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

You may also run updates as queries against a set of models:

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

To delete a model, simply call the `delete` method on the instance:

Deleting An Existing Model

```
$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();
```

If you wish to simply update the timestamps on a model, you may use the `touch` method:

Updating Only The Model's Timestamps

```
$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, specify the `softDelete` property on the model:

```
class User extends Eloquent {

    protected $softDelete = true;

}
```

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. To force soft deleted models to appear in a result set, use the `withTrashed` method on the query:

Forcing Soft Deleted Models Into Results

```
$users = User::withTrashed()->where('account_id', 1)->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();
```

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 Eloquent {

    protected $table = 'users';

    public $timestamps = false;

}
```

If you wish to customize the format of your timestamps, you may override the `getDateFormat` method in your model:

Providing A Custom Timestamp Format

```
class User extends Eloquent {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

Query Scopes

Scopes allow you to easily re-use query logic in your models. To define a scope, simply prefix a model method with `scope`:

Defining A Query Scope

```
class User extends Eloquent {

    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 Eloquent {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

Then pass the parameter into the scope call:

```
$users = User::ofType('member')->get();
```

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 four types of relationships:

- One To One
- One To Many
- Many To Many
- Polymorphic Relations

One To One

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:

Defining A One To One Relation

```
class User extends Eloquent {

    public function phone()
    {
        return $this->hasOne('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:

```
return $this->hasOne('Phone', 'custom_key');
```

To define the inverse of the relationship on the `Phone` model, we use the `belongsTo` method:

Defining The Inverse Of A Relation

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('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 Eloquent {

    public function user()
    {
        return $this->belongsTo('User', 'custom_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 Eloquent {

    public function comments()
    {
        return $this->hasMany('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:

```
return $this->hasMany('Comment', 'custom_key');
```

To define the inverse of the relationship on the `Comment` model, we use the `belongsTo` method:

Defining The Inverse Of A Relation

```
class Comment extends Eloquent {

    public function post()
    {
        return $this->belongsTo('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 Eloquent {

    public function roles()
    {
        return $this->belongsToMany('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('Role', 'user_roles');
```

You may also override the conventional associated keys:

```
return $this->belongsToMany('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 Eloquent {

    public function users()
    {
        return $this->belongsToMany('User');
    }

}
```

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 Eloquent {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}

class Order extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

Now, we can retrieve the photos for either a staff member or an order:

Retrieving A Polymorphic Relation

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

However, the true "polymorphic" magic is when you access the staff or order from the `Photo` model:

Retrieving The Owner Of A Polymorphic Relation

```
$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.

To help understand how this works, let's explore the database structure for a polymorphic relation:

Polymorphic Relation Table Structure

```
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.

Querying Relations

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:

Checking Relations When Selecting

```
$posts = Post::has('comments')->get();
```

You may also specify an operator and a count:

```
$posts = Post::has('comments', '>=', 3)->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 Eloquent {

    public function user()
    {
        return $this->belongsTo('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;
```

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 Eloquent {

    public function author()
    {
        return $this->belongsTo('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".

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

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:

Attaching A Related Model

```
$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.

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);
```

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:

Using Sync To Attach Many To Many Models

```
$user->roles()->sync(array(1, 2, 3));
```

You may also associate other pivot table values with the given IDs:

Adding Pivot Data When Syncing

```
$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 Eloquent {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('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('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('Role')->withTimestamps();
```

To delete all records on the pivot table for a model, you may use the `detach` method:

Deleting Records On A Pivot Table

```
User::find(1)->roles()->detach();
```

Note that this operation does not delete records from the `roles` table, but only from the pivot table.

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.

For example, we may determine if a result set contains a given primary key using the `contains` method:

Checking If A Collection Contains A Key

```
$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;
```

Eloquent collections also contain a few helpful methods for looping and filtering the items they contain:

Iterating & Filtering Collections

```
$roles = $user->roles->each(function($role)
{

});

$roles = $user->roles->filter(function($role)
{

});
```

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;
});
```

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:

Returning A Custom Collection Type

```
class User extends Eloquent {

    public function newCollection(array $models = array())
    {
        return new CustomCollection($models);
    }

}
```

Accessors & Mutators

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:

Defining An Accessor

```
class User extends Eloquent {

    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.

Mutators are declared in a similar fashion:

Defining A Mutator

```
class User extends Eloquent {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

Date Mutators

By default, Eloquent will convert the `created_at`, `updated_at`, and `deleted_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 timetamp, 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();
}
```

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.

If `false` is returned from the `creating`, `updating`, `saving`, or `deleting` events, the action will be cancelled:

Cancelling Save Operations Via Events

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

Eloquent models also contain a static `boot` method, which may provide a convenient place to register your event bindings.

Setting A Model Boot Method

```
class User extends Eloquent {

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

        // Setup event bindings...
    }

}
```

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

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:

Converting A Model To An Array

```
$user = User::with('roles')->first();

return $user->toArray();
```

Note that entire collections of models may also be converted to arrays:

```
return User::all()->toArray();
```

To convert a model to JSON, you may use the `toJson` method:

Converting A Model To JSON

```
return User::find(1)->toJson();
```

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!

Returning A Model From A Route

```
Route::get('users', function()
{
    return User::all();
});
```

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:

Hiding Attributes From Array Or JSON Conversion

```
class User extends Eloquent {

    protected $hidden = array('password');

}
```

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: {#eloquent-array-appends}

```
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.

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
`$table->increments('id');`	Incrementing ID to the table (primary key).
`$table->bigIncrements('id');`	Incrementing ID using a "big integer" equivalent.
`$table->string('email');`	VARCHAR equivalent column
`$table->string('name', 100);`	VARCHAR equivalent with a length
`$table->integer('votes');`	INTEGER equivalent to the table
`$table->bigInteger('votes');`	BIGINT equivalent to the table
`$table->smallInteger('votes');`	SMALLINT equivalent to the table
`$table->float('amount');`	FLOAT equivalent to the table
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale
`$table->boolean('confirmed');`	BOOLEAN equivalent to the table
`$table->date('created_at');`	DATE equivalent to the table
`$table->dateTime('created_at');`	DATETIME 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->softDeletes();`	Adds **deleted_at** column for soft deletes
`$table->text('description');`	TEXT equivalent to the table
`$table->binary('data');`	BLOB equivalent to the table
`$table->enum('choices', array('foo', 'bar'));`	ENUM equivalent to the table
`->nullable()`	Designate that the column allows NULL values
`->default($value)`	Declare a default value for a column
`->unsigned()`	Set INTEGER to UNSIGNED

If you are using the MySQL database, you may use the `after` method to specify the order of columns:

Using After On MySQL

```
$table->string('name')->after('email');
```

Renaming Columns

To rename a column, you may use the `renameColumn` method on the Schema builder:

Renaming A Column

```
Schema::table('users', function($table)
{
    $table->renameColumn('from', 'to');
});
```

> **Note:** Renaming `enum` column types is not supported.

Dropping Columns

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('votes', 'avatar', 'location');
});
```

Checking Existence

You may easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

Checking For Existence Of Table

```
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:

Fluently Creating A Column And Index

```
$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:

Adding A Foreign Key To A Table

```
$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.

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

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 scheme.

Creating Migrations

To create a migration, you may use the `migrate:make` command on the Artisan CLI:

Creating A Migration

```
php artisan migrate:make create_users_table
```

The migration will be placed in your `app/database/migrations` folder, and will contain a timestamp which allows the framework to determine the order of the migrations.

You may also specify a `--path` option when creating the migration. The path should be relative to the root directory of your installation:

```
php artisan migrate:make foo --path=app/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 migrate:make create_users_table --table=users --create
```

Running Migrations

Running All Outstanding Migrations

```
php artisan migrate
```

Running All Outstanding Migrations For A Path

```
php artisan migrate --path=app/foo/migrations
```

Running All Outstanding Migrations For A Package

```
php artisan migrate --package=vendor/package
```

> **Note:** If you receive a "class not found" error when running migrations, try running the `composer update` command.

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 `app/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
```

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.

> **Note:** If you have the Redis PHP extension installed via PECL, you will need to rename the alias for Redis in your `app/config/app.php` file.

Configuration

The Redis configuration for your application is stored in the **app/config/database.php** file. Within this file, you will see a **redis** array containing the Redis servers used by your application:

```
'redis' => array(

    'cluster' => true,

    'default' => array('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

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

To view a list of all available Artisan commands, you may use the `list` command:

Listing All Available Commands

```
php artisan list
```

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

Viewing The Help Screen For A Command

```
php artisan help migrate
```

You may specify the configuration environment that should be used while running a command using the `--env` switch:

Specifying The Configuration Environment

```
php artisan migrate --env=local
```

You may also view the current version of your Laravel installation using the `--version` option:

Displaying Your Current Laravel Version

```
php artisan --version
```

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/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 `command:make` Artisan command, which will generate a command stub to help you get started:

Generate A New Command Class

```
php artisan command:make FooCommand
```

By default, generated commands will be stored in the `app/commands` directory; however, you may specify custom path or namespace:

```
php artisan command:make FooCommand --path=app/classes --namespace=Classes
```

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);
```

Registering Commands

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/start/artisan.php` file. Within this file, you may use the `Artisan::add` method to register the command:

Registering An Artisan Command

```
Artisan::add(new CustomCommand);
```

If your command is registered in the application IoC container, you may use the `Artisan::resolve` method to make it available to Artisan:

Registering A Command That Is In The IoC Container

```
Artisan::resolve('binding.name');
```

Calling Other Commands

Sometimes you may wish to call other commands from your command. You may do so using the `call` method:

Calling Another Command

```
$this->call('command.name', array('argument' => 'foo', '--option' => 'bar'));
```