LEARNING

Advance Laravel

Free Contribution with university Sandford USA

www.techprofree.com

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: laravel

It is an unofficial and free Laravel ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Laravel.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Laravel

Remarks

Laravel StackOverflow Slack Community

Coming soon

Featured Tutorial

Getting Started With Laravel

Contribution Guidelines

Coming soon

Contribution Style Guide

Coming soon

About Laravel

Created by Taylor Otwell as a free open-source PHP web framework, Laravel is meant to ease and accelerate the development process of web applications with a great taste for simplicity.

It follows the model–view–controller (MVC) architectural pattern as well as the PSR-2 coding standard, and the PSR-4 autoloading standard.

Running a Test Driven Development (TDD) in Laravel is fun and easy to implement.

Hosted on GitHub and available at https://github.com/laravel/laravel, Laravel boasts of a micro-services architecture, making it tremendously extendable and this, with ease, with the use of custom-made and or existing third-party packages.

Main Features

MVC

Laravel uses the MVC model, therefore there are three core-parts of the framework which work

together: models, views and controllers. Controllers are the main part where most of the work is done. They connect to models to get, create or update data and display the results on views, which contain the actual HTML structure of the application.

Blade Templating Engine

Laravel is shipped with a templating engine known as Blade. Blade is quite easy to use, yet, powerful. One feature the Blade templating engine does not share with other popular ones is her permissiveness; allowing the use of plain PHP code in Blade templating engine files.

It is important to note that Blade templating engine files have `.blade` appended to file names right before the usual `.php` which is nothing other than the actual file extension. As such, `.blade.php` is the resulting file extension for Blade template files. Blade template engine files are stored in the resources/views directory.

Routing & Middleware

You can define the URLs of your application with the help of routes. These routes can contain variable data, connect to controllers or can be wrapped into middlewares. Middelware is a mechanism for filtering HTTP requests. They can be used to interact with requests before they reach the controllers and can thus modify or reject requests.

Artisan

Artisan is the command line tool you can use to control parts of Laravel. There are a lot of commands available to create models, controllers and other resources needed for development. You can also write your own commands to extend the Artisan command line tool.

Eloquent ORM

To connect your models to various types of databases, Laravel offers its own ORM with a large set of functions to work with. The framework also provides migration and seeding and also features rollbacks.

Event Handling

The framework is capable of handling events across the application. You can create event listeners and event handlers that are similar to the ones from NodeJs.

Versions

Version	Release Date
1.0	2011-06-09
2.0	2011-11-24

Version	Release Date
3.0	2012-02-22
3.1	2012-03-27
3.2	2012-05-22
4.0	2013-05-28
4.1	2013-12-12
4.2	2014-06-01
5.0	2015-02-04
5.1 (LTS)	2015-06-09
5.2	2015-12-21
5.3	2016-08-24
5.4	2017-01-24

Examples

Welcome to Laravel tag documentation!

Laravel is a well-known PHP Framework. Here, you will learn all-about Laravel. Starting from *as-simple-as* knowing what Object-Oriented Programming is, to the advanced Laravel package development topic.

This, like every other Stackoverflow documentation tag, is community-driven documentation, so if you already have experiences on Laravel, share your knowledge by add your own topics or examples! Just don't forget to consult our **Contribution style guide** on this topic remarks to know more about how to contribute and the style guide that we made to make sure we can give the best experience towards people that want to learn more about Laravel.

More than that, we are very glad that you come, hope we can see you often here!

Starter Guide

Starter guide is custom navigation that we ordered by ourselves to make topic browsing easier especially for beginner. This navigation is ordered by level of difficulty.

Getting Started

Installation

Laravel Views

Blade : Introduction

Blade : Variables and Control Structures

Or

Installation from here

1. Get composer from here and install it
2. Get Wamp from here, install it and set environment variable of PHP
3. Get path to `www` and type command:

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

To install a specific Laravel version, get path to `www` and type command:

```
composer create-project --prefer-dist laravel/laravel=DESIRED_VERSION projectname
```

Or

Via Laravel Installer

First, download the Laravel installer using Composer:

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

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

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

```
laravel new blog
```

Read Getting started with Laravel online: https://riptutorial.com/laravel/topic/794/getting-started-with-laravel

Chapter 2: Artisan

Syntax

- php artisan [command] [options] [arguments]

Parameters

Command	Description
clear-compiled	Remove the compiled class file
down	Put the application into maintenance mode
env	Display the current framework environment
help	Displays help for a command
list	Lists commands
migrate	Run the database migrations
optimize	Optimize the framework for better performance
serve	Serve the application on the PHP development server
tinker	Interact with your application
up	Bring the application out of maintenance mode
app:name	Set the application namespace
auth:clear-resets	Flush expired password reset tokens
cache:clear	Flush the application cache
cache:table	Create a migration for the cache database table
config:cache	Create a cache file for faster configuration loading
config:clear	Remove the configuration cache file
db:seed	Seed the database with records
event:generate	Generate the missing events and listeners based on registration
key:generate	Set the application key

Command	Description
make:auth	Scaffold basic login and registration views and routes
make:console	Create a new Artisan command
make:controller	Create a new controller class
make:event	Create a new event class
make:job	Create a new job class
make:listener	Create a new event listener class
make:middleware	Create a new middleware class
make:migration	Create a new migration file
make:model	Create a new Eloquent model class
make:policy	Create a new policy class
make:provider	Create a new service provider class
make:request	Create a new form request class
make:seeder	Create a new seeder class
make:test	Create a new test class
migrate:install	Create the migration repository
migrate:refresh	Reset and re-run all migrations
migrate:reset	Rollback all database migrations
migrate:rollback	Rollback the last database migration
migrate:status	Show the status of each migration
queue:failed	List all of the failed queue jobs
queue:failed-table	Create a migration for the failed queue jobs database table
queue:flush	Flush all of the failed queue jobs
queue:forget	Delete a failed queue job
queue:listen	Listen to a given queue
queue:restart	Restart queue worker daemons after their current job

Command	Description
queue:retry	Retry a failed queue job
queue:table	Create a migration for the queue jobs database table
queue:work	Process the next job on a queue
route:cache	Create a route cache file for faster route registration
route:clear	Remove the route cache file
route:list	List all registered routes
schedule:run	Run the scheduled commands
session:table	Create a migration for the session database table
vendor:publish	Publish any publishable assets from vendor packages
view:clear	Clear all compiled view files

Examples

Introduction

Artisan is a utility that can help you do specific repetitive tasks with bash commands. It covers many tasks, including: working with database **migrations** and **seeding**, clearing **cache**, creating necessary files for **Authentication** setup, **making** new *controllers, models, event classes*, and a lot more.

> 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.

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

```
php artisan list
```

To know more about the any available command, just precede its name with **help** keyword:

```
php artisan help [command-name]
```

List all registered routes filtered by multiple methods

```
php artisan route:list --method=GET --method=POST
```

This will include all routes that accept `GET` and `POST`methods simultaneously.

Running Laravel Artisan commands using PHP code

You can also use Laravel Artisan commands from your routes or controllers.

To run a command using PHP code:

```
Artisan::call('command-name');
```

For example,

```
Artisan::call('db:seed');
```

Creating and registering new artisan command

You can create new commands via

```
php artisan make:command [commandName]
```

So this will create [commandName] command class inside `app/Console/Commands`directory. inside

this class you will find `protected $signature` and `protected $description`variables, it represents name and discription of your command which will be used to describe your command.

after creating command you can register your command inside `app/Console/Kernel.php` class where you will find `commands` property.

so you can add your command inside the $command array like :

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

and then i can use my command via console.

so as example i have named my command like

```
protected $signature = 'test:command';
```

So whenever i will run

```
php artisan test:command
```

it will call the `handle` method inside the class having signature `test:command`.

Read Artisan online: https://riptutorial.com/laravel/topic/1140/artisan

Chapter 3: Authentication

Examples

Multi Authentication

Laravel allows you to use multiple Authentication types with specific guards.

In laravel 5.3 multiple authentication is little different from Laravel 5.2

I will explain how to implement multiauthentication feature in 5.3

First you need two different user Model

```
cp App/User.php App/Admin.php
```

change class name to Admin and set namespace if you use models different. it should look like

App\Admin.php

```
<?php namespace

App;

use Illuminate\Foundation\Auth\User as Authenticatable; use
Illuminate\Notifications\Notifiable;

class Admin extends Authenticatable
{
        use Notifiable;

        protected $fillable = ['name', 'email', 'password']; protected $hidden     =
        ['password', 'remember_token'];
```

Also you need create a migration for admin

```
php artisan make:migration create_admins_table
```

then edit migration file with contents of default user migration. Looks like this

```
<?php

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

class CreateAdminsTable extends Migration
{
```

```
     *  Run the migrations.
     *
     *  @return void
     */
    public function up()
    {
        Schema::create('admins', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();

            $table->softDeletes();
        });
    }

    /**
     *  Reverse the migrations.
     *
     *  @return void
     */
    public function down()
    {
```

edit **config/auth.php**

```
'guards'        => [ 'web'
                => [
                'driver'        => 'session',
                'provider' => 'users',
        ],

        'api'       => [
                'driver'        => 'token',
                'provider' => 'users',
        ],
        //Add Admin Guard
        'admin' => [
                'driver'        => 'session',
                'provider' => 'admins',
```

and

```
'providers' => [
        'users'     => [
                'driver' => 'eloquent', 'model' =>
                App\User::class,
        ],
        //Add Admins Provider
        'admins' => [
                'driver' => 'eloquent', 'model' =>
```

```
            ],
        ],
```

Notice that we add two entry. one in **guards** variable one in **providers** variable.

And this is how you use the other guard then "web"

My App\Http\Controllers\Admin\LoginController

```
<?php

namespace App\Http\Controllers\Admin; use

App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers; use
Illuminate\Support\Facades\Auth;

class AuthController extends Controller
{

        use AuthenticatesUsers; protected

        $guard = 'admin';

        protected $redirectTo = '/admin/';

        public function showLoginForm()
        {
                return view('admin.login');
        }

        protected function guard()
        {
                return Auth::guard($this->guard);
```

this needs little explanation.

in a nutshell **Auth::guard('admin')** will allow you to use auth methods (such as login, logout, register etc.) with your admin guard.

For example

```
Auth::guard('admin')->login($user)
```

will search $user in admins table and login with the user while

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

will works normally with users table. Default guard is specified in **config/auth.php** with *defaults* array. In fresh laravel it is "web" .

In controller you have to implement methods from AuthenticatesUsers to show your custom view paths. And you need implement other functions such as guard to use your new user guards.

In this example my admin login is **admin/login.blade**

And by implementing guard() function to return **Auth::guard('admin')** all AuthenticatesUsers trait methods works with "admin" guard.

In earlier versions of laravel, this is little different from 5.3

in 5.2 getGuard function returns $guard variable from class and main function (login) use it in

```
Auth::guard($guard)->attempt(...)
```

in 5.3 guard function returns whole Auth::guard() and main function use it like

```
$this->guard()->attempt(...)
```

Read Authentication online: https://riptutorial.com/laravel/topic/7051/authentication

Chapter 4: Authorization

Introduction

Laravel provides a simple way to authorise user actions on specific resources. With Authorization, you can selectively allow users access to certain resources while denying access to others. Laravel provides a simple API for managing user authorizations by using `Gates` and `Policies`. `Gates` provide a simple closure based approach to authorisation using the `AuthServiceProvider` while `Policies` allow you to organise authorisation logic around models using classes.

Examples

Using Gates

`Gates` are closures that determine if a user is allowed to perform a certain action on a resource. `Gates` are typically defined in the boot method of `AuthServiceProvider` and succinctly named to reflect what it's doing. An example of a gate that allows only premium users to view some content will look like this:

```
Gate::define('view-content', function ($user, $content){ return $user-
        >isSubscribedTo($content->id);
});
```

A `Gate` always receives a user instance as the first argument, you don't need to pass it when using the gate, and may optionally receive additional arguments such as the eloquent model in concern.

Authorizing Actions with Gates

To use the example above on a blade template to hide content from the user, you would typically do something like this:

```
@can('view-content', $content)
  <! -- content here --> @endcan
```

To completely prevent navigation to the content, you can do the following in your controller:

```
if(Gate::allows('view-content', $content)){
        /* user can view the content */
}

                OR

if(Gate::denies('view-content', $content)){
        /* user cannot view content */
```

Note: You are not required to pass the currently authenticated user to these method, Laravel takes care of that for you.

Policies

Policies are classes that help you organise authorisation logic around a model resource. Using our previous example, we might have a `ContentPolicy` that manages user access to the `Content`model.

To make `ContentPolicy`, laravel provides an artisan command. Simply run

```
php artisan make:policy ContentPolicy
```

This will make an empty policy class and place in `app/Policies` folder. If the folder does not exist, Laravel will create it and place the class inside.

Once created, policies need to be registered to help Laravel know which policies to use when authorising actions on models. Laravel's `AuthServiceProvider`, which comes with all fresh Laravel installations, has a policies property which maps your eloquent models to their authorisation policies. All you need to do add the mapping to the array.

```
protected $policies = [
        Content::class => ContentPolicy::class,
  ];
```

Writing Policies

Writing `Policies` follows much the same pattern as writing `Gates`. The content permission gate can be rewritten as a Policy like this:

```
function view($user, $content)
{
        return $user->isSubscribedTo($content->id);
}
```

Policies can contain more methods as needed to take care of all authorisation cases for a model.

Authorizing Actions with Policies

Via The User model

The Laravel User model contains two methods that help with authorisations using `Policies`; `can` and `can't`. These two can be used to determine if a user has authorisation or not on a model respectively.

To check if a user can view a content or not, you can do the following:

```
if($user->can('view', $content)){
        /* user can view content */
}
```

```
            OR

                if($user->cant('view', $content)){
                  /* user cannot view content */
}
```

Via Middleware

```
             Route::get('/contents/{id}, function(Content $content){
            /* user can view content */
        })->middleware('can:view,content');
```

Via Controllers

Laravel provides a helper method, called `authorize` that takes the name of the policy and the associated model as arguments, and either authorizes the action based on your authorisation logic or denies the action and throws an `AuthorizationException` which the Laravel Exception handler converts to a `403 HTTP response`.

```
pubic function show($id)
{
        $content = Content::find($id);

        $this->authorize('view', $content);

        /* user can view content */
```

Read Authorization online: https://riptutorial.com/laravel/topic/9360/authorization

Chapter 5: Blade Templates

Introduction

Laravel supports Blade templating engine out of the box. The Blade templating engine allows us to create master templates and child templating loading content from master templates, we can have variables, loops and conditional statements inside the blade file.

Examples

Views: Introduction

Views, in an MVC pattern, contain the logic on *how* to present data to the user. In a web application, typically they are used to generate the HTML output that is sent back to users with each response. By default, views in Laravel are stored in the `resources/views` directory.

A view can be called using the `view` helper function:

```
view(string $path, array $data = [])
```

The first parameter of the helper is the path to a view file, and the second parameter is an optional array of data to pass to the view.

Therefore, to call the `resources/views/example.php`, you would use:

```
view('example');
```

View files in subfolders within the `resources/views` directory, such as `resources/views/parts/header/navigation.php`, can be called using dot notation:
`view('parts.header.navigation');`

Within a view file, such as `resources/views/example.php`, you're free to include both HTML and PHP together:

```
<html>
    <head>
        <title>Hello world!</title>
    </head>
    <body>
        <h1>Welcome!</h1>
        <p>Your name is: <?php echo $name; ?></p>
    </body>
```

In the previous example (which doesn't use any Blade specific syntax), we output the `$name` variable. To pass this value to our view, we would pass an array of values when calling the view helper:

```
view('example', ['name' => $name]);
```

or alternatively, use the compact()helper. In this case, the string passed to compact() corresponds to the name of the variable we want to pass to the view.

```
view('example', compact('name'));
```

NAMING CONVENTION FOR BLADE VARIABLES

While sending data back to view. You can use underscore for multi-words variablebut with -laravel gives error.

Like this one will give error (notice hyphen (-) within the user-address

```
view('example',['user-address' => 'Some Address']);
```

The **correct way** of doing this will be

```
view('example', ['user_address' => 'Some Address']);
```

Control Structures

Blade provides convenient syntax for common PHP control structures.

Each of the control structures begins with @[structure] and ends with @[endstructure]. Notice that within the tags, we are just typing normal HTML and including variables with the Blade syntax.

Conditionals

'If' statements

```
@if ($i > 10)
        <p>{{ $i }} is large.</p> @elseif ($i ==
10)
        <p>{{ $i }} is ten.</p> @else
        <p>{{ $i }} is small.</p> @endif
```

'Unless' statements

(Short syntax for 'if not'.)

```
@unless ($user->hasName())
        <p>A user has no name.</p>
@endunless
```

Loops

'While' loop

```
@while (true)
    <p>I'm looping forever.</p> @endwhile
```

'Foreach' loop

```
@foreach ($users as $id => $name)
    <p>User {{ $name }} has ID {{ $id }}.</p> @endforeach
```

'Forelse' Loop

(Same as 'foreach' loop, but adds a special `@empty` directive, which is executed when the array expression iterated over is empty, as a way to show default content .)

```
@forelse($posts as $post)
    <p>{{ $post }} is the post content.</p> @empty
    <p>There are no posts.</p>
@endforelse
```

Within loops, a special `$loop` variable will be available, containing information about the state of the loop:

Property	Description
$loop->index	The index of the current loop iteration (starts at 0).
$loop->iteration	The current loop iteration (starts at 1).
$loop->remaining	The remaining loop iterations.
$loop->count	The total number of items in the array being iterated.
$loop->first	Whether this is the first iteration through the loop.
$loop->last	Whether this is the last iteration through the loop.
$loop->depth	The nesting level of the current loop.
$loop->parent	When in a nested loop, the parent's loop variable.

Example:

```
@foreach ($users as $user) @foreach ($user-
    >posts as $post)
            @if ($loop->parent->first)
                This is first iteration of the parent loop. @endif
        @endforeach
@endforeach
```

Since Laravel 5.2.22, we can also use the directives `@continue` and `@break`

Property	Description
`@continue`	Stop the current iteration and start the next one.
`@break`	Stop the current loop.

Example :

```
@foreach ($users as $user) @continue
        ($user->id == 2)
            <p>{{ $user->id }} {{ $user->name }}</p> @break ($user->id
        == 4)
@endforeach
```

Then (*assuming 5+ users are sorted by ID and no ID is missing*) the page will render

```
1 Dave
3  John
4  William
```

Echoing PHP expressions

Any PHP expression within double curly braces `{{ $variable }}` will be `echo`ed after being run through the `e` helper function. (So html special characters (`<`, `>`, `"`, `'`, `&`) are safely replaced for the corresponding html entities.) (The PHP expression must evaluate to string, otherwise an exception will be thrown.)

Echoing a variable

```
{{ $variable }}
```

Echoing an element in an array

```
{{ $array["key"] }}
```

Echoing an object property

```
{{ $object->property }}
```

Echoing the result of a function call

```
{{ strtolower($variable) }}
```

Checking for Existence

Normally, in PHP, to check if a variable is set and print it you would do

- Before PHP 7

```
<?php echo isset($variable) ? $variable : 'Default'; ?>
```

- After PHP 7 (using the "Null coalescing operator")

```
<?php echo $variable ?? 'Default'; ?>
```

Blade operator `or` makes this easier:

```
{{ $variable or 'Default' }}
```

Raw echos

As mentioned, regular double braces syntax `{{ }}`, are filtered through PHP's `htmlspecialchars` function, for security (preventing malicious injection of HTML in the view). If you would like to bypass this behavior, for example if you're trying to output a block of HTML content resulting from a PHP expression, use the following syntax:

```
{!! $myHtmlString !!}
```

Note that it is considered a best practice to use the standard `{{ }}` syntax to escape your data, unless absolutely necessary. In addition, when echoing untrusted content (ie. content supplied by users of your site), you should avoid using the `{!! !!}` syntax.

Including Partial Views

With Blade, you can also include partial views (called 'partials') directly into a page like so:

```
@include('includes.info', ['title' => 'Information Station'])
```

The code above will include the view at 'views/includes/info.blade.php'. It will also pass in a variable `$title` having value 'Information Station'.

In general, an included page will have access to any variable that the calling page has access to. For instance, if we have:

```
{{$user}} // Outputs 'abc123'
@include('includes.info')
```

And 'includes/info.blade.php' has the following:

```
<p>{{$user}} is the current user.</p>
```

Then the page will render:

```
abc123
abc123 is the current user.
```

Include Each

Sometimes, you will want to combine an `include` statement with a `foreach`statement, and access the variables from within the foreach loop in the include. In this case, use Blade's `@each`directive:

```
@each('includes.job', $jobs, 'job')
```

The first parameter is the page to include. The second parameter is the array to iterate over. The third parameter is the variable assigned to the elements of the array. The statement above is equivalent to:

```
@foreach($jobs as $job) @include('includes.job', ['job' => $job])
@endforeach
```

You can also pass an optional fourth argument to the `@each` directive to specify the view to show when the array is empty.

```
@each('includes.job', $jobs, 'job', 'includes.jobsEmpty')
```

Layout Inheritance

A layout is a view file, which is extended by other views which inject blocks of code into their parent. For example:

parent.blade.php:

```
<html>
    <head>
        <style type='text/css'>
```

```
        @yield('styling')
        </style>
    </head>
    <body>
        <div class='main'>
        @yield('main-content')
        </div>
    </body>
```

child.blade.php:

```
@extends('parent')

@section('styling')
.main {
    color: red;
}
@stop

@section('main-content') This is
child page! @stop
```

otherpage.blade.php:

```
@extends('parent')

@section('styling')
.main {
    color: blue;
}
@stop

@section('main-content') This is
another page! @stop
```

Here you see two example child pages, which each extend the parent. The child pages define a `@section`, which is inserted in the parent at the appropriate `@yield` statement.

So the view rendered by `View::make('child')` will say "**This is child page!**" in red, while `View::make('otherpage')` will produce the same html, except with the text "**This is another page!**" in blue instead.

It is common to separate the view files, e.g. having a layouts folder specifically for the layout files, and a separate folder for the various specific individual views.

The layouts are intended to apply code that should appear on every page, e.g. adding a sidebar or header, without having to write out all the html boilerplate in every individual view.

Views can be extended repeatedly - i.e. **page3** can `@extend('page2')`, and **page2** can `@extend('page1')`.

The extend command uses the same syntax as used for `View::make` and `@include`, so the file `layouts/main/page.blade.php` is accessed as `layouts.main.page`.

Sharing data to all views

Sometimes you need to set the same data in many of your views.

Using View::share

```
// "View" is the View Facade
View::share('shareddata', $data);
```

After this, the contents of `$data` will be available in all views under the name `$shareddata`.

`View::share` is typically called in a service provider, or perhaps in the constructor of a controller, so the data will be shared in views returned by that controller only.

Using View::composer

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location. You can directly bind variable to a specific view or to all views.

Closure-based composer

```
use Illuminate\Support\Facades\View;

// ...

View::composer('*', function ($view) {
        $view->with('somedata', $data);
});
```

Class-based composer

```
use Illuminate\Support\Facades\View;

// ...

View::composer('*', 'App\Http\ViewComposers\SomeComposer');
```

As with `View::share`, it's best to register the composers in a service provider.

If going with the composer class approach, then you would have `App/Http/ViewComposers/SomeComposer.php` with:

```
use Illuminate\Contracts\View\View;

class SomeComposer
{
        public function compose(View $view)
        {
                $view->with('somedata', $data);
        }
```

These examples use `'*'` in the composer registration. This parameter is a string that matches the view names for which to register the composer (`*` being a wildcard). You can also select a single view (e.g. `'home'`) of a group of routes under a subfolder (e.g. `'users.*'`).

Execute arbitrary PHP code

Although it might not be proper to do such thing in a view if you intend to separate concerns strictly, the `php`Blade directive allows a way to execute PHP code, for instance, to set a variable:

```
@php($varName = 'Enter content ')
```

(same as:)

```
@php
        $varName = 'Enter content '; @endphp
```

later:

```
{{ $varName }}
```

Result:

Enter content

Read Blade Templates online: https://riptutorial.com/laravel/topic/1407/blade-templates

Chapter 6: Cashier

Remarks

Laravel Cashier can be used for subscription billing by providing an interface into the subscription services of both Braintree and Stripe. In addition to basic subscription management it can be used to handle coupons, exchanging subscriptions, quantities, cancellation grace periods and PDF invoice generation.

Examples

Stripe Setup

Initial Setup

To use Stripe for handling payments we need to add the following to the `composer.json` then run `composer update`:

```
"laravel/cashier": "~6.0"
```

The following line then needs to be added to `config/app.php`, the service provider:

```
Laravel\Cashier\CashierServiceProvider
```

Databse Setup

In order to use cashier we need to configure the databases, if a users table does not already exist we need to create one and we also need to create a subscriptions table. The following example amends an existing `users` table. See Eloquent Models for more information about models.

To use cashier create a new migration and add the following which will achieve the above:

```
// Adjust users table

Schema::table('users', function ($table) {
        $table->string('stripe_id')->nullable();
        $table->string('card_brand')->nullable();
        $table->string('card_last_four')->nullable();
        $table->timestamp('trial_ends_at')->nullable();
});

//Create subscriptions table Schema::create('subscriptions', function

($table) {
        $table->increments('id');
        $table->integer('user_id');
        $table->string('name');
```

```
        $table->string('stripe_plan');
        $table->integer('quantity');
        $table->timestamp('trial_ends_at')->nullable();
        $table->timestamp('ends_at')->nullable();
        $table->timestamps();
});
```

We then need to run `php artisan migrate` to update our database.

Model Setup

We then have to add the billable trait to the User model found in `app/User.php` and change it to the following:

```
use Laravel\Cashier\Billable;

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

Stripe Keys

In order to ensure that we ares ending the money to our own Stripe account we have to set it up in the `config/services.php` file by adding the following line:

```
'stripe' => [
        'model'         => App\User::class, 'secret' =>
        env('STRIPE_SECRET'),
],
```

Replacing the `STRIPE_SECRET` with your own stripe secret key.

After completing this Cashier and Strip is setup so you can continue with setting up subscriptions.

Read Cashier online: https://riptutorial.com/laravel/topic/7474/cashier

Chapter 7: Change default routing behaviour in Laravel 5.2.31 +

Syntax

- public function map(Router $router) // Define the routes for the application.
- protected function mapWebRoutes(Router $router) // Define the "web" routes for the application.

Parameters

Parameter	Header
Router $router	\Illuminate\Routing\Router $router

Remarks

Middleware means that every call to a route will go through the middleware before actually hitting your route specific code. In Laravel the web middleware is used to ensure session handling or the csrf token check for example.

There are other middlewares like auth or api by default. You can also easily create your own middleware.

Examples

Adding api-routes with other middleware and keep default web middleware

Since Laravel version 5.2.31 the web middleware is applied by default within the RouteServiceProvider (https://github.com/laravel/laravel/commit/5c30c98db96459b4cc878d085490e4677b0b67ed)

In app/Providers/RouteServiceProvider.php you will find the following functions which apply the middleware on every route within your app/Http/routes.php

```
public function map(Router $router)
{
        $this->mapWebRoutes($router);
}

// ...

protected function mapWebRoutes(Router $router)
```

```
        $router->group([
                'namespace' => $this->namespace, 'middleware' => 'web',
        ], function ($router) {
                require app_path('Http/routes.php');
        });
}
```

As you can see the **middleware** web is applied. You could change this here. However, you can also easily add another entry to be able to put your api routes for example into another file (e.g. routes-api.php)

```
public function map(Router $router)
{
        $this->mapWebRoutes($router);
        $this->mapApiRoutes($router);
}

protected function mapWebRoutes(Router $router)
{
        $router->group([
                'namespace' => $this->namespace, 'middleware' => 'web',
        ], function ($router) {
                require app_path('Http/routes.php');
        });
}

protected function mapApiRoutes(Router $router)
{
        $router->group([
                'namespace' => $this->namespace, 'middleware' => 'api',
        ], function ($router) {
```

With this you can easily seperate you api routes from your application routes without the messy group wrapper within your routes.php

Read Change default routing behaviour in Laravel 5.2.31 + online: https://riptutorial.com/laravel/topic/4285/change-default-routing-behaviour-in-laravel-5-2-31-plus

Chapter 8: Collections

Syntax

- $collection = collect(['Value1', 'Value2', 'Value3']); // Keys default to 0, 1, 2, ...,

Remarks

`Illuminate\Support\Collection` provides a fluent and convenient interface to deal with arrays of data. You may well have used these without knowing, for instance Model queries that fetch multiple records return an instance of `Illuminate\Support\Collection`.

For up to date documentation on Collections you can find the official documentation here

Examples

Creating Collections

Using the `collect()` helper, you can easily create new collection instances by passing in an array such as:

```
$fruits = collect(['oranges', 'peaches', 'pears']);
```

If you don't want to use helper functions, you can create a new Collection using the class directly:

```
$fruits = new Illuminate\Support\Collection(['oranges', 'peaches', 'pears']);
```

As mentioned in the remarks, Models by default return a `Collection` instance, however you are free to create your own collections as needed. If no array is specified on creation, an empty Collection will be created.

where()

You can select certain items out of a collection by using the `where()` method.

```
$data = [
    ['name' => 'Taylor',                'coffee_drinker' => true], ['name' =>
    'Matt', 'coffee_drinker' => true]
];

$matt = collect($data)->where('name', 'Matt');
```

This bit of code will select all items from the collection where the name is 'Matt'. In this case, only the second item is returned.

Nesting

Just like most array methods in Laravel, `where()` supports searching for nested elements as well. Let's extend the example above by adding a second array:

```
$data = [
        ['name' => 'Taylor',                    'coffee_drinker' => ['at_work' => true, 'at_home' => true]], ['name' => 'Matt',
        'coffee_drinker' => ['at_work' => true, 'at_home' => false]]
];

$coffeeDrinkerAtHome = collect($data)->where('coffee_drinker.at_home', true);
```

This will only return Taylor, as he drinks coffee at home. As you can see, nesting is supported using the dot-notation.

Additions

When creating a Collection of objects instead of arrays, those can be filtered using `where()` as well. The Collection will then try to receive all desired properties.

5.3

Please note, that since Laravel 5.3 the `where()` method will try to loosely compare the values by default. That means when searching for `(int)1`, all entries containing `'1'` will be returned as well. If you don't like that behaviour, you may use the `whereStrict()` method.

Using Get to lookup value or return default

You often find yourself in a situation where you need to find a variables corresponding value, and collections got you covered.

In the example below we got three different locales in an array with a corresponding calling code assigned. We want to be able to provide a locale and in return get the associated calling code. The second parameter in `get` is a default parameter if the first parameter is not found.

```
function lookupCallingCode($locale)
{
        return collect([ 'de_DE' => 49,
                'en_GB' => 44,
                'en_US' => 1,
        ])->get($locale, 44);
}
```

In the above example we can do the following

```
lookupCallingCode('de_DE'); // Will return 49
lookupCallingCode('sv_SE'); // Will return 44
```

You may even pass a callback as the default value. The result of the callback will be returned if the specified key does not exist:

```
    return collect([ 'de_DE' => 49,
        'en_GB' => 44,
        'en_US' => 1,
    ])->get($locale, function() { return 44;
    });
```

Using Contains to check if a collection satisfies certain condition

A common problem is having a collection of items that all need to meet a certain criteria. In the example below we have collected two items for a diet plan and we want to check that the diet doesn't contain any unhealthy food.

```
// First we create a collection
$diet = collect([
    ['name' => 'Banana', 'calories' => '89'], ['name' => 'Chocolate',
    'calories' => '546']
]);

// Then we check the collection for items with more than 100 calories
$isUnhealthy = $diet->contains(function ($i, $snack) { return $snack["calories"]
    >= 100;
```

In the above case the `$isUnhealthy`variable will be set to `true`as Chocolate meets the condition, and the diet is thus unhealthy.

Using Pluck to extract certain values from a collection

You will often find yourself with a collection of data where you are only interested in parts of the data.

In the example below we got a list of participants at an event and we want to provide a the tour guide with a simple list of names.

```
// First we collect the participants
$participants = collect([
    ['name' => 'John', 'age' => 55],
    ['name' => 'Melissa', 'age' => 18],
    ['name' => 'Bob', 'age' => 43],
    ['name' => 'Sara', 'age' => 18],
]);

// Then we ask the collection to fetch all the names
$namesList = $partcipants->pluck('name')
```

You can also use `pluck` for collections of objects or nested arrays/objects with dot notation.

```
$users = User::all(); // Returns Eloquent Collection of all users
$usernames = $users->pluck('username'); // Collection contains only user names

$users->load('profile'); // Load a relationship for all models in collection

// Using dot notation, we can traverse nested properties
$names = $users->pluck('profile.first_name'); // Get all first names from all user profiles
```

Using Map to manipulate each element in a collection

Often you need to change the way a set of data is structured and manipulate certain values.

In the example below we got a collection of books with an attached discount amount. But we much rather have a list of books with a price that's already discounted.

```
$books = [
        ['title' => 'The Pragmatic Programmer', 'price' => 20, 'discount' => 0.5], ['title' => 'Continuous Delivery', 'price'
        => 25, 'discount' => 0.1], ['title' => 'The Clean Coder', 'price' => 10, 'discount' => 0.75],
];

$discountedItems =              collect($books)->map(function ($book) {
      return ['title' => $book["title"], 'price' => $book["price"] * $book["discount"]];
});

//[
//          ['title' => 'The Pragmatic Programmer', 'price' => 10],
//          ['title' => 'Continuous Delivery', 'price' => 12.5],
//          ['title' => 'The Clean Coder', 'price' => 5],
```

This could also be used to change the keys, let's say we wanted to change the key `title` to `name` this would be a suitable solution.

Using sum, avg, min or max on a collection for statistical calculations

Collections also provide you with an easy way to do simple statistical calculations.

```
$books = [
        ['title' => 'The Pragmatic Programmer', 'price' => 20], ['title' => 'Continuous
        Delivery', 'price' => 30], ['title' => 'The Clean Coder', 'price' => 10],
]

$min = collect($books)->min('price'); // 10
$max = collect($books)->max('price'); // 30
$avg = collect($books)->avg('price'); // 20
$sum = collect($books)->sum('price'); // 60
```

Sorting a collection

There are a several different ways of sorting a collection.

Sort()

The sort method sorts the collection:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort(); echo $sorted-

>values()->all(); returns : [1, 2, 3, 4, 5]
```

The sort method also allows for passing in a custom callback with your own algorithm. Under the hood sort uses php's usort.

```
$collection = $collection->sort(function ($a, $b) { if ($a == $b) {
            return 0;
    }
    return ($a < $b) ? -1 : 1;
});
```

SortBy()

The sortBy method sorts the collection by the given key:

```
$collection = collect([
['name' => 'Desk', 'price' => 200],
['name' => 'Chair', 'price' => 100], ['name' => 'Bookcase',
'price' => 150],
]);

$sorted = $collection->sortBy('price'); echo $sorted-

>values()->all();

returns:        [
            ['name' => 'Chair', 'price' => 100], ['name' => 'Bookcase',
            'price' => 150], ['name' => 'Desk', 'price' => 200],
```

The sortBy method allows using dot notation format to access deeper key in order to sort a multi-dimensional array.

```
$collection = collect([
    ["id"=>1,"product"=>['name' => 'Desk', 'price' => 200]],
    ["id"=>2, "product"=>['name' => 'Chair', 'price' => 100]],
    ["id"=>3, "product"=>['name' => 'Bookcase', 'price' => 150]],
    ]);
```

```
$sorted = $collection->sortBy("product.price")->toArray();

return: [
    ["id"=>2, "product"=>['name' => 'Chair', 'price' => 100]],
    ["id"=>3, "product"=>['name' => 'Bookcase', 'price' => 150]],
    ["id"=>1,"product"=>['name' => 'Desk', 'price' => 200]],
]
```

SortByDesc()

This method has the same signature as the `sortBy` method, but will sort the collection in the opposite order.

Using reduce()

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration. Please see reduce method.

The `reduce` method loops through each item with a collection and produces new result to the next iteration. Each result from the last iteration is passed through the first parameter (in the following examples, as `$carry`).

This method can do a lot of processing on large data sets. For example the following examples, we will use the following example student data:

```
 $student = [
      ['class' => 'Math', 'score' => 60],
      ['class' => 'English', 'score' => 61], ['class' => 'Chemistry',
      'score' => 50], ['class' => 'Physics', 'score' => 49],
];
```

Sum student's total score

```
$sum = collect($student)
      ->reduce(function($carry, $item){ return $carry +
            $item["score"];
      }, 0);
```

Result: `220`

Explanation:

- `$carry` is the result from the last iteration.
- The second parameter is the default value for the $carry in the first round of iteration. This case, the default value is 0

Pass a student if all their scores are >= 50

```
$isPass = collect($student)
```

```
    ->reduce(function($carry, $item){
        return $carry && $item["score"] >= 50;
    }, true);
```

Result: false

Explanation:

- Default value of $carry is true
- If all score is greater than 50, the result will return true; if any less than 50, return false.

Fail a student if any score is < 50

```
$isFail = collect($student)
    ->reduce(function($carry, $item){
        return $carry || $item["score"] < 50;
    }, false);
```

Result: true

Explain:

- the default value of $carry is false
- if any score is less than 50, return true; if all scores are greater than 50, return false.

Return subject with the highest score

```
$highestSubject = collect($student)
    ->reduce(function($carry, $item){
        return $carry === null || $item["score"] > $carry["score"] ? $item : $carry;
    });
```

result: ["subject" => "English", "score" => 61]

Explain:

- The second parameter is not provided in this case.
- The default value of $carry is null, thus we check for that in our conditional.

Using macro() to extend collections

The macro() function allows you to add new functionality to Illuminate\Support\Collectionobjects Usage:

```
Collection::macro("macro_name", function ($parameters) {
    // Your macro
});
```

For example:

```
Collection::macro('uppercase', function () { return $this-
        >map(function ($item) {
                return strtoupper($item);
        });
});
```

Result: ["HELLO", "WORLD"]

Using Array Syntax

The Collectionobject implements the ArrayAccessand IteratorAggregateinterface, allowing it to be used like an array.

Access collection element:

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

Result: 2

Assign new element:

```
$collection = collect([1, 2, 3]);
$collection[] = 4;
```

Result: $collection is [1, 2, 3, 4]

Loop collection:

```
$collection = collect(["a" => "one", "b" => "two"]);
$result = "";
foreach($collection as $key => $value){
        $result .= "(".$key.": ".$value.") ";
}
```

Result: $result is (a: one) (b: two)

Array to Collection conversion:

To convert a collection to a native PHP array, use:

```
$array = $collection->all();
//or
$array = $collection->toArray()
```

To convert an array into a collection, use:

```
$collection = collect($array);
```

Using Collections with Array Functions

Please be aware that collections are normal objects which won't be converted properly when used by functions explicitly requiring arrays, like `array_map($callback)`.

Be sure to convert the collection first, or, if available, use the method provided by the `Collection` class instead: `$collection->map($callback)`

Read Collections online: https://riptutorial.com/laravel/topic/2358/collections

Chapter 9: Common Issues & Quick Fixes

Introduction

This section lists the common issues & quick fixes developers (especially beginners) face.

Examples

TokenMisMatch Exception

You get this exception mostly with form submissions. Laravel protects application from `CSRF` and validates every request and ensures the request originated from within the application. This validation is done using a `token`. If this token mismatches this exception is generated.

Quick Fix

Add this within your form element. This sends `csrf_token` generated by laravel along with other form data so laravel knows that your request is valid

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

Read Common Issues & Quick Fixes online: https://riptutorial.com/laravel/topic/9971/common-issues---quick-fixes

Chapter 10: Constants

Examples

Example

First you have to create a file constants.php and it is a good practice to create this file inside app/config/ folder. You can also add constants.php file in compose.json file.

Example File:

app/config/constants.php

Array based constants inside the file:

```
return [
    'CONSTANT' => 'This is my first constant.'
];
```

And you can get this constant by including the facade `Config` :

```
use Illuminate\Support\Facades\Config;
```

Then get the value by constant name `CONSTANT` like below :

```
echo Config::get('constants.CONSTANT');
```

And the result would be the value :

This is my first constant.

Read Constants online: https://riptutorial.com/laravel/topic/9192/constants

Chapter 11: Controllers

Introduction

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

Examples

Basic Controllers

```
<?php

namespace App\Http\Controllers; use

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

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

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

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

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

Controller Middleware

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

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

However, it is more convenient to specify middleware within your controller's constructor. Using the middleware method from your controller's constructor, you may easily assign middleware to the controller's action.

```
class UserController extends Controller
{
        /**
          *   Instantiate a new controller instance.
          *
          *   @return void
          */
        public function_construct()
        {
                $this->middleware('auth');

                $this->middleware('log')->only('index');

                $this->middleware('subscribed')->except('store');
```

Resource Controller

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

```
php artisan make:controller PhotoController --resource
```

This command will generate a controller at `app/Http/Controllers/PhotoController.php`. The controller will contain a method for each of the available resource operations.

Example of how a Resource Controller look

```
<?php

namespace App\Http\Controllers; use

Illuminate\Http\Request;

class PhotoController extends Controller
{
        /**
          *   Display a listing of the resource.
          *
          *   @return \Illuminate\Http\Response
          */
        public function index()
        {
                //
        }

        /**
          *   Show the form for creating a new resource.
          *
          *   @return \Illuminate\Http\Response
```

```
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param     \Illuminate\Http\Request                $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param     int       $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param     int       $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param     \Illuminate\Http\Request                $request
     * @param     int       $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param     int       $id
     * @return \Illuminate\Http\Response
     */
```

```
    public function destroy($id)
    {
        //
    }
}
```

The example of the resource controller shares the method name of those in the table below.

Next, you may register a resourceful route to the controller:

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

This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions, including notes informing you of the HTTP verbs and URIs they handle.

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Read Controllers online: https://riptutorial.com/laravel/topic/10604/controllers

Chapter 12: Cron basics

Introduction

Cron is a task scheduler daemon which runs scheduled tasks at certain intervals. Cron uses a configuration file called crontab, also known as cron table, to manage the scheduling process.

Examples

Create Cron Job

Crontab contains cron jobs, each related to a specific task. Cron jobs are composed of two parts, the cron expression, and a shell command to be run:

```
* * * * * command/to/run
```

Each field in the above expression `* * * * *` is an option for setting the schedule frequency. It is composed of minute, hour, day of month, month and day of week in order of the placement. The asterisk symbol refers to all possible values for the respective field. As a result, the above cron job will be run every minute in the day.

The following cron job is executed at **12:30** every day:

```
30 12 * * * command/to/run
```

Read Cron basics online: https://riptutorial.com/laravel/topic/9891/cron-basics

Chapter 13: Cross Domain Request

Examples

Introduction

Sometimes we need cross domain request for our API's in laravel. We need to add appropriate headers to complete the cross domain request successfully. So we need to make sure that whatever headers we are adding should be accurate otherwise our API's become vulnerable. In order to add headers we need to add middleware in laravel which will add the appropriate headers and forward the requests.

CorsHeaders

```
<?php

namespace laravel\Http\Middleware; class

CorsHeaders
{
    /**
     *  This must be executed _before_ the controller action since _after_ middleware isn't executed when exceptions are thrown
and caught by global handlers.
     *
     *  @param $request
     *  @param \Closure $next
     *  @param string [$checkWhitelist] true or false Is a string b/c of the way the arguments are supplied.
     *  @return mixed
     */
    public function handle($request, \Closure $next, $checkWhitelist = 'true')
    {
        if ($checkWhitelist == 'true') {
            // Make sure the request origin domain matches one of ours before sending CORS response headers.
            $origin = $request->header('Origin');
            $matches = [];
            preg_match('/^(https?:\/\/)?([a-zA-Z\d]+\.)*(?<domain>[a-zA-Z\d-\.]+\.[a-z]{2,10})$/',
$origin, $matches);

            if (isset($matches['domain']) && in_array($matches['domain'], ['yoursite.com']) { header('Access-Control-Allow-Origin: '
                . $origin);
                header('Access-Control-Expose-Headers: Location'); header('Access-
                Control-Allow-Credentials: true');

                // If a preflight request comes then add appropriate headers if ($request->method() ===
                'OPTIONS') {
                    header('Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH'); header('Access-Control-Allow-
                    Headers: ' . $request->header('Access-Control-Request-
Headers'));
                        // 20 days
```

```
        } else {
            header('Access-Control-Allow-Origin: *');
        }

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

Read Cross Domain Request online: https://riptutorial.com/laravel/topic/7425/cross-domain-request

Chapter 14: Custom Helper function

Introduction

Adding custom helpers can assist you with your development speed. There are a few things to take into consideration while writing such helper functions though, hence this tutorial.

Remarks

Just a few pointers:

- We've put the function definitions within a check (`function_exists`) to prevent exceptions when the service provider is called twice.
- An alternative way is registering the helpers file from the `composer.json` file. You can copy the logic from the laravel framework itself.

Examples

document.php

```
<?php

if (!function_exists('document')) { function
        document($text = '') {
                return $text;
        }
```

Create a helpers.php file, let's assume for now it lives in `app/Helpers/document.php`. You can put many helpers in one file (this is how Laravel does it) or you can split them up by name.

HelpersServiceProvider.php

Now let's create a service provider. Let's put it under `app/Providers`:

```
<?php

namespace App\Providers;

class HelpersServiceProvider extends ServiceProvider
{
        public function register()
        {
                require_once __DIR__ . '/../Helpers/document.php';
        }
```

The above service provider load the helpers file and registers your custom function automatically.

Please make sure you register this HelpersServiceProvider in your `config/app.php` under `providers`:

```
'providers' => [
        // [..] other providers
        App\Providers\HelpersServiceProvider::class,
]
```

Use

Now you can use the function `document()`everywhere in your code, for example in blade templates. This example only returns the same string it receives as an argument

```
<?php
Route::get('document/{text}', function($text) { return
      document($text);
});
```

Now go to `/document/foo` in your browser (use `php artisan serve`or `valet`), which will return `foo`.

Read Custom Helper function online: https://riptutorial.com/laravel/topic/8347/custom-helper-function

Chapter 15: CustomException class in Laravel

Introduction

PHP Exceptions are thrown when an unprecedented event or error occurs.

As a rule of thumb, an exception should not be used to control the application logic such as if-statements and should be a subclass of the Exception class.

One main advantage of having all exceptions caught by a single class is that we are able to create custom exception handlers that return different response messages depending on the exception.

Examples

CustomException class in laravel

all errors and exceptions, both custom and default, are handled by the Handler class in app/Exceptions/Handler.php with the help of two methods.

- report()
- render()

```
public function render($request, Exception $e)
{
  //check if exception is an instance of ModelNotFoundException. if ($e instanceof
  ModelNotFoundException)
  {
        // ajax 404 json feedback if
        ($request->ajax())
        {
              return response()->json(['error' => 'Not Found'], 404);
        }
        // normal 404 view page feedback
        return response()->view('errors.missing', [], 404);
    }
```

then create view related to error in errors folder named 404.blade.php

User not found.

You broke the balance of the internet

Read CustomException class in Laravel online:
https://riptutorial.com/laravel/topic/9550/customexception-class-in-laravel

Chapter 16: Database

Examples

Multiple database connections

Laravel allows user work on multiple database connections. If you need to connect to multiple databases and make them work together, you are beware of the connection setup.

You also allow using different types of database in the same application if you required.

Default connection In config/database.php, you can see the configuration item call:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

This name references the connections' name mysql below:

```
'connections' => [

        'sqlite' => [
                'driver' => 'sqlite',
                'database' =>              database_path('database.sqlite'), 'prefix' => '',
        ],

        'mysql' => [
                'driver' => 'mysql',
                'host' => env('DB_HOST', 'localhost'), 'port' =>
                env('DB_PORT', '3306'),
                'database' => env('DB_DATABASE', 'forge'), 'username' =>
                env('DB_USERNAME', 'forge'), 'password' =>
                env('DB_PASSWORD', ''), 'charset' => 'utf8',
                'collation' => 'utf8_unicode_ci', 'prefix' => '',
                'strict' => false, 'engine' =>
                null,
        ],
],
```

If you did not mention the name of database connection in other codes or commands, Laravel will pick up the default database connection name. however, in multiple database connections, even you setup the default connection, you've better setup everywhere which database connection you used.

Migration file

In migration file, if single database connection, you can use:

```
Schema::create("table",function(Blueprint $table){
        $table->increments('id');
```

```
});
```

In multiple database connection, you will use the `connection()` method to tell Laravel which database connection you use:

```
  Schema::connection("sqlite")->create("table",function(Blueprint $table){
        $table->increments('id');
});
```

Artisan Migrate

if you use single database connection, you will run:

```
php artisan migrate
```

However, for multiple database connection, you've better tell which database connection maintains the migration data. so you will run the following command:

```
php artisan migrate:install --database=sqlite
```

This command will install migration table in the target database to prepare migration.

```
php artisan migrate --database=sqlite
```

This command will run migration and save the migration data in the target database

```
php artisan migrate:rollback --database=sqlite
```

This command will rollback migration and save the migration data in the target database

Eloquent Model

To specify a database connection using Eloquent, you need to define the `$connection` property:

```
namespace App\Model\Sqlite; class
Table extends Model
{
        protected $table="table"; protected
        $connection = 'sqlite';
}
```

To specify another (second) database connection using Eloquent:

```
namespace App\Model\MySql; class
Table extends Model
{
        protected $table="table"; protected
        $connection = 'mysql';
}
```

Laravel will use `$connection` property defined in a model to utilize the specified connection defined in `config/database.php`. If the `$connection` property is not defined in a model the default will be used.

You may also specify another connection using the static `on` method:

```
// Using the sqlite connection Table::on('sqlite')-
>select(...)->get()
// Using the mysql connection Table::on('mysql')-
>select(...)->get()
```

Database/Query Builder

You may also specify another connection using the query builder:

```
// Using the sqlite connection
DB::connection('sqlite')->table('table')->select(...)->get()
// Using the mysql connection
DB::connection('mysql')->table('table')->select(...)->get()
```

Unit Test

Laravel provide `seeInDatabase($table,$fielsArray,$connection)` to test database connection code. In Unit test file, you need to do like:

```
$this
        ->json(
                'GET',
                'result1/2015-05-08/2015-08-08/a/123'
        )
          ->seeInDatabase("log", ["field"=>"value"], 'sqlite');
```

In this way, Laravel will know which database connection to test.

Database Transactions in Unit Test

Laravel allows database to rollback all the change during the tests. For testing multiple database connections, you need to set `$connectionsToTransact` properties

```
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends TestCase
{
          use DatabaseTransactions;

          $connectionsToTransact =["mysql","sqlite"] //tell Laravel which database need to rollBack public function testExampleIndex()
        {
                $this->visit('/action/parameter')
                  ->see('items');
        }
}
```

Read Database online: https://riptutorial.com/laravel/topic/1093/database

Chapter 17: Database Migrations

Examples

Migrations

To control your database in Laravel is by using migrations. Create migration with artisan:

```
php artisan make:migration create_first_table --create=first_table
```

This will generate the class CreateFirstTable. Inside the up method you can create your columns:

```
<?php

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

class CreateFirstTable extends Migration
{
        public function up()
        {
                Schema::create('first_table', function (Blueprint $table) {
                        $table->increments('id');
                        $table->string('first_string_column_name');
                        $table->integer('secont_integer_column_name');
                        $table->timestamps();
                });
        }

        public function down()
        {
```

At the end to run all of your migrations classes you can run the artisan command:

```
php artisan migrate
```

This will create your tables and your columns in your database. Other useful migrate command are:

- `php artisan migrate:rollback` - Rollback the last database migration
- `php artisan migrate:reset` - Rollback all database migrations
- `php artisan migrate:refresh` - Reset and re-run all migrations
- `php artisan migrate:status` - Show the status of each migration

Modifying existing tables

Sometimes, you need to change your existing table structure like `renaming/deleting` columns.

Which you can accomplish by creating a new migration.And In the `up` method of your migration.

```
//Renaming Column.

public function up()
{
	Schema::table('users', function (Blueprint $table) {
		$table->renameColumn('email', 'username');
	});
}
```

Above example will rename `email column` of `users table`to `username`. While the below code drops a column `username` from `users`table.

IMPROTANT : For modifying columns you need to add `doctrine/dbal` dependency to project's `composer.json` file and run `composer update`to reflect changes.

```
//Droping Column public
function up()
{
	Schema::table('users', function (Blueprint $table) {
		$table->dropColumn('username');
	});
}
```

The migration files

Migrations in a Laravel 5 application live in the `database/migrations`directory. Their filenames conform to a particular format:

```
<year>_<month>_<day>_<hour><minute><second>_<name>.php
```

One migration file should represent a schema update to solve a particular problem. For example:

```
2016_07_21_134310_add_last_logged_in_to_users_table.php
```

Database migrations are kept in chronological order so that Laravel knows in which order to execute them. Laravel will always execute migrations from oldest to newest.

Generating migration files

Creating a new migration file with the correct filename every time you need to change your schema would be a chore. Thankfully, Laravel's `artisan` command can generate the migration for you:

```
php artisan make:migration add_last_logged_in_to_users_table
```

You can also use the `--table` and `--create`flags with the above command. These are optional and just for convenience, and will insert the relevant boilerplate code into the migration file.

```
php artisan make:migration add_last_logged_in_to_users_table --table=users

php artisan make:migration create_logs_table --create=logs
```

You can specify a custom output path for the generated migration using the `--path` option. The path is relative to the application's base path.

```
php artisan make:migration --path=app/Modules/User/Migrations
```

Inside a database migration

Each migration should have an `up()`method and a `down()`method. The purpose of the `up()`method is to perform the required operations to put the database schema in its new state, and the purpose of the `down()` method is to reverse any operations performed by the `up()`method. Ensuring that the `down()` method correctly reverses your operations is critical to being able to rollback database schema changes.

An example migration file may look like this:

```
<?php

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

class AddLastLoggedInToUsersTable extends Migration
{
        /**
          *  Run the migrations.
          *
          *  @return void
          */
        public function up()
        {
                Schema::table('users', function (Blueprint $table) {
                        $table->dateTime('last_logged_in')->nullable();
                });
        }

        /**
          *  Reverse the migrations.
          *
          *  @return void
          */
        public function down()
        {
                Schema::table('users', function (Blueprint $table) {
```

When running this migration, Laravel will generate the following SQL to run against your database:

```
ALTER TABLE `users` ADD `last_logged_in` DATETIME NULL
```

Running migrations

Once your migration is written, running it will apply the operations to your database.

```
php artisan migrate
```

If all went well, you'll see an output similar to the below:

```
Migrated: 2016_07_21_134310_add_last_logged_in_to_users_table
```

Laravel is clever enough to know when you're running migrations in the production environment. If it detects that you're performing a destructive migration (for example, one that removes a column from a table), the `php artisan migrate` command will ask you for confirmation. In continuous delivery environments this may not be wanted. In that case, use the `--force` flag to skip the confirmation:

```
php artisan migrate --force
```

If you've only just run migrations, you may be confused to see the presence of a `migrations` table in your database. This table is what Laravel uses to keep track of what migrations have already been run. When issuing the `migrate` command, Laravel will determine what migrations have yet to run, and then execute them in chronological order, and then update the `migrations` table to suit.

You should never manually edit the `migrations` table unless you absolutely know what you're doing. It's very easy to inadvertently leave your database in a broken state where your migrations will fail.

Rolling Back Migrations

What if you want to rollback the latest migration i.e recent operation, you can use the awesome `rollback` command. But remember that this command rolls back only the last migration, which may include multiple migration files

```
php artisan migrate:rollback
```

If you are interested in rolling back all of your application migrations, you may use the following command

```
php artisan migrate:reset
```

Moreover if you are lazy like me and want to rollback and migrate with one command, you may use this command

```
php artisan migrate:refresh
php artisan migrate:refresh --seed
```

You can also specify number of steps to rollback with `step` option. Like this will rollback 1 step.

```
php artisan migrate:rollback --step=1
```

Read Database Migrations online: https://riptutorial.com/laravel/topic/1131/database-migrations

Chapter 18: Database Seeding

Examples

Running a Seeder

You may add your new Seeder to the DatabaseSeeder class.

```
/**
  *  Run the database seeds.
  *
  *  @return void
  */
public function run()
{
        $this->call(UserTableSeeder::class);
```

To run a database seeder, use the Artisan command

```
php artisan db:seed
```

This will run the DatabaseSeeder class. You can also choose to use the `--class=` option to manually specify which seeder to run.

*Note, you may need to run composer dumpautoload if your Seeder class cannot be found. This typically happens if you manually create a seeder class instead of using the artisan command.

Creating a Seed

Database seeds are stored in the /database/seeds directory. You can create a seed using an Artisan command.

```
php artisan make:seed UserTableSeeder
```

Alternatively you can create a new class which extends `Illuminate\Database\Seeder`. The class must a public function named `run()`.

Inserting Data using a Seeder

You can reference models in a seeder.

```
use DB;
use App\Models\User;

class UserTableSeeder extends Illuminate\Database\Seeder{ public function run(){
```

```
            # Remove all existing entrie
            DB::table('users')->delete() ; User::create([
                    'name' => 'Admin',
                    'email' => 'admin@example.com', 'password' =>
                    Hash::make('password')
            ]);

    }
}
```

Inserting data with a Model Factory

You may wish to use Model Factories within your seeds. This will create 3 new users.

```
use App\Models\User;

class UserTableSeeder extends Illuminate\Database\Seeder{ public function run(){
            factory(User::class)->times(3)->create();
    }
}
```

You may also want to define specific fields on your seeding like a password, for instance. This will create 3 users with the same password.

```
factory(User::class)->times(3)->create(['password' => '123456']);
```

Seeding with MySQL Dump

Follow previous example of creating a seed. This example uses a MySQL Dump to seed a table in the project database. The table must be created before seeding.

```
<?php

use Illuminate\Database\Seeder; class

UserTableSeeder extends Seeder
{

    /**
     *  Run the database seeds.
     *
     *  @return void
     */
     public function run()
     {
            $sql = file_get_contents(database_path() . '/seeds/users.sql');

            DB::statement($sql);
```

Our $sql is going to be the contents of our users.sql dump. The dump should have an INSERT INTO statement. It will be up to you where you store your dumps. In the above example, it is stored in the project directory `\database\seeds`. Using laravel's helper function `database_path()` and appending the directory and file name of the dump.

```
INSERT INTO `users` (`id`, `name`, `email`, `password`, `remember_token`, `created_at`,
`updated_at`) VALUES
(1, 'Jane', 'janeDoe@fakemail.com', 'superSecret', NULL, '2016-07-21 00:00:00', '2016-07-21
00:00:00'),
(2, 'John', 'johnny@fakemail.com', 'sup3rS3cr3t', NULL, '2016-07-21 00:00:00', '2016-07-21
00:00:00');
```

`DB::statement($sql)` will execute the inserts once the Seeder is run. As in previous examples, you can put the `UserTableSeeder` in the `DatabaseSeeder`class provided by laravel:

```
<?php

use Illuminate\Database\Seeder; class

DatabaseSeeder extends Seeder
{
        /**
          *   Run the database seeds.
          *
          *   @return void
          */
        public function run()
        {
                $this->call(UserTableSeeder::class);
        }
```

and run from CLI in project directory `php artisan db:seed`. Or you can run the Seeder for a single class using `php artisan db:seed --class=UsersTableSeeder`

Using faker And ModelFactories to generate Seeds

1) BASIC SIMPLE WAY

Database-driven applications often need data pre-seeded into the system for testing and demo purposes.

To make such data, first create the seeder class

ProductTableSeeder

```
use Faker\Factory as Faker; use
App\Product;

class ProductTableSeeder extends DatabaseSeeder { public function

run()
{
        $faker = $this->getFaker();
```

```
        for ($i = 0; $i < 10; $i++)
        {
                $name =                 $faker->word;
                $image =                $faker->imageUrl;

                Modelname::create([ 'name' =>
                        $name, 'image' => $image,
                ]);
            }
        }
  }
```

To call a be able to execute a seeder class, you have call it from the DatabaseSeeder class, Simply by passing the name of the seeder you wish to run:

use Illuminate\Database\Seeder;

```
class DatabaseSeeder extends Seeder { protected

        $faker;

        public function getFaker() { if (empty($this-
                >faker)) {
                        $faker = Faker\Factory::create();
                        $faker->addProvider(new Faker\Provider\Base($faker));
                        $faker->addProvider(new Faker\Provider\Lorem($faker));
                }
                return $this->faker = $faker;
        }
        public function run() {
                $this->call(ProductTableSeeder::class);
```

Do not forget to run `$ composer dump-autoload` after you create the Seeder, since they are not automatically autoloaded by composer (unless you created seeder by artisan command `$ php artisan make:seeder Name`)

Now you are ready to seed by running this artisan command `php artisan db:seed`

2) USING Model Factories

First of all you to define a default set of attributes for each Model in App/database/factories/ModelFactory.php

Taking a User model as an exemple, This how a ModelFactory looks like

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

```
});
```

Now Create a table seeder `php artisan make:seeder UsersTableSeeder`

And add this

```
public function run()
{
        factory(App\User::class, 100)->create()
}
```

then add this to the `DatabaseSeeder`

```
public function run()
{
        $this->call(UsersTableSeeder::class);
}
```

This will seed the table with 100 records.

Read Database Seeding online: https://riptutorial.com/laravel/topic/1118/database-seeding

Chapter 19: Deploy Laravel 5 App on Shared Hosting on Linux Server

Remarks

To get more information on deploying Laravel project on shared hosting, visit this Github repo.

Examples

Laravel 5 App on Shared Hosting on Linux Server

By default Laravel project's `public` folder exposes the content of the app which can be requested from anywhere by anyone, the rest of the app code is invisible or inaccessible to anyone without proper permissions.

After developing the application on your development machine, it needs to be pushed to a production server so that it can be accessed through the internet from anywhere - right?

For most apps/websites the first choice is to use shared hosting package from hosting service providers like GoDaddy, HostGator etc. mainly due to low cost.

> **note**: you may ask your provider to manually change **document_root**, so all you have to do is upload your Laravel application to server (via FTP), request change of root to **{app}/public** and you should be good.

Such shared hosting packages, however do have limitations in terms of terminal access and file permissions. By default one has to upload their app/code to the `public_html` folder on their shared hosting account.

So if you want to upload a Laravel project to a shared hosting account how would you go about it? Should you upload the entire app (folder) to the `public_html` folder on your shared hosting account? - **Certainly NO**

Because everything in the `public_html` folder is accessible "publically i.e. by anyone" which would be a big security risk.

Steps to upload a project to shared hosting account - the Laravel way

Step 1
Create a folder called laravel (or anything you like) on the same level as the `public_html` folder.

```
Eg:
/
|--var
      |---www                          //create this folder in your shared hosting account
```

```
| ------- public_html
| ------- log
```

Step 2
Copy every thing except the `public` folder from your laravel project (on development machine) in the `laravel` folder (on server host - shared hosting account).
You can use:

- C-panel : which would be the slowest option
- FTP Client: like ***FileZilla*** to connect to you shared hosting account and transfer your files and folders through FTP upload
- Map Network Drive: you can also create a mapped network drive on your development machine to connect to your shared hosting account's root folder using "ftp://your-domain-name" as the network address.

Step 3
Open the `public` folder of your laravel project (on development machine), copy everything and paste in the `public_html` folder (on server host - shared hosting account).
Step 4
Now open the `index.php` file in the `public_html`folder on the shared hosting account (in cpanel editor or any other connected editor) and:

Change:

```
require_DIR__.'/../bootstrap/autoload.php';
```

To:

```
require_DIR__.'/../laravel/bootstrap/autoload.php';
```

And Change:

```
$app = require_once_DIR_.'/../bootstrap/app.php';
```

To:

```
$app = require_once_DIR_.'/../laravel/bootstrap/app.php';
```

Save and close.

Step 5
Now go to the `laravel` folder (on shared hosting account -server) and open `server.php`file
Change

```
require_once_DIR_.'/public/index.php';
```

To:

```
require_once DIR_.'../public_html/index.php';
```

Save and close.

Step 6
Set file permissions for the `laravel/storage` folder (recursively) and all files, sub-folders and file within them on shared hosting account - server to `777`.
Note: Be careful with the file permissions in linux, they are like double edged sword, if not used correctly, they may make your app vulnerable to attacks. For understanding Linux file permissions you can read https://www.linux.com/learn/tutorials/309527-understanding-linux-file-permissions

Step 7

As `.env` file of local/development server is Ignored by git and it should be ignored as it has all the environment variables including the APP_KEY and it should not be exposed to public by pushing it into the repositories'. You can also see that `.gitignore` file has `.env` mentioned thus it will not upload it to repositories.

After following all the above steps make a `.env`file in the laravel folder and add all the environment variable which you have used from the local/development server's `.env` file to the `.env` file of production server.

Even there are configuration files like `app.php`, `database.php` in config folder of laravel application which defines this variables as by default in second parameter of `env()` but don't hard-code the values in these files as it will affect the configuration files of the users who pulls your repository. So it is recommended to create `.env` file manually!

Also laravel gives `.env-example`file that you can use as a reference.

That's it.

Now when you visit the url which you configured as the domain with your server, your laravel app should work just as it worked on your localhost - development machine, while still the application code is safe and not accessible by anyone without proper file permissions.

Read Deploy Laravel 5 App on Shared Hosting on Linux Server online:
https://riptutorial.com/laravel/topic/2410/deploy-laravel-5-app-on-shared-hosting-on-linux-server

Chapter 20: Directory Structure

Examples

Change default app directory

There are use cases when you might want to rename your app directory to something else. In Laravel4 you could just change a config entry, here's one way to do it in Laravel5.

In this example we'll be renaming the `app` directory to `src`.

Override Application class

The directories name `app`is hardcoded into the core Application class, so it has to be overridden. Create a new file `Application.php`. I prefer to keep mine in the `src` directory (the one we'll be replacing app with), but you can place it elsewhere.

Here's how the overridden class should look like. If you want a different name, just change the string `src` to something else.

```
namespace App;

class Application extends \Illuminate\Foundation\Application
{
        /**
          * @inheritdoc
          */
        public function path($path = '')
        {
                return $this->basePath . DIRECTORY_SEPARATOR . 'src' . ($path ? DIRECTORY_SEPARATOR .
$path : $path);
        }
```

Save the file. We're done with it.

Calling the new class

Open up `bootstrap/app.php` and locate

```
$app = new Illuminate\Foundation\Application( realpath(__DIR__
        .'/../')
);
```

We'll be replacing it with this

```
$app = new App\Application( realpath(__DIR__.'/../')
```

```
);
```

Composer

Open up your `composer.json` file and change autoloading to match your new location

```
"psr-4": {
        "App\\": "src/"
}
```

And finally, in the command line run `composer dump-autoload` and your app should be served from the `src` directory.

Change the Controllers directory

if we want to change the `Controllers` directory we need:

1. Move and/or rename the default `Controllers` directory where we want it. For example from `app/Http/Controllers` to `app/Controllers`
2. Update all the namespaces of the files inside the `Controllers` folder, making they adhere to the new path, respecting the PSR-4 specific.
3. Change the namespace that is applied to the `routes.php` file, by editing `app\Providers\RouteServiceProvider.php` and change this:

```
protected $namespace = 'App\Http\Controllers';
```

to this:

```
protected $namespace = 'App\Controllers';
```

Read Directory Structure online: https://riptutorial.com/laravel/topic/3153/directory-structure

Chapter 21: Eloquent

Introduction

The Eloquent is an ORM (Object Relational Model) included with the Laravel. It implements the active record pattern and is used to interact with relational databases.

Remarks

Table naming

The convention is to use pluralised “snake_case” for table names and singular “StudlyCase” for model names. For example:

- A `cats` table would have a `Cat`model
- A `jungle_cats` table would have a `JungleCat`model
- A `users` table would have a `User`model
- A `people` table would have a `Person`model

Eloquent will automatically try to bind your model with a table that has the plural of the name of the model, as stated above.

You can, however, specify a table name to override the default convention.

```
class User extends Model
{
        protected $table = 'customers';
}
```

Examples

Introduction

Eloquent is the ORM built into the Laravel framework. It allows you to interact with your database tables in an object-oriented manner, by use of the ActiveRecord pattern.

A single model class usually maps to a single database table, and also relationships of different types (one-to-one, one-to-many, many-to-many, polymorphic) can be defined between different model classes.

Section Making a Model describes the creation and definition of model classes.

Before you can start using Eloquent models, make sure at least one database connection has been configured in your `config/database.php` configuration file.

To understand usage of eloquent query builder during development you may use `php artisan ide-`

helper:generate command. Here is the link.

Sub-topic Navigation

Eloquent Relationship

Persisting

In addition to reading data with Eloquent, you can also use it to insert or update data with the save() method. If you have created a new model instance then the record will be *inserted*; otherwise, if you have retrieved a model from the database and set new values, it will be *updated*.

In this example we create a new User record:

```
$user = new User();
$user->first_name = 'John';
$user->last_name = 'Doe';
$user->email = 'john.doe@example.com';
$user->password = bcrypt('my_password');
$user->save();
```

You can also use the create method to populate fields using an array of data:

```
User::create([ 'first_name'=> 'John',
        'last_name' => 'Doe',
        'email'             =>'john.doe@example.com',
        'password'          => bcrypt('changeme'),
]);
```

When using the create method your attributes should be declared in the fillable array within your model:

```
class User extends Model
{
        protected $fillable = [
                'first_name', 'last_name',
                'email', 'password',
        ];
}
```

Alternatively, if you would like to make all attributes mass assignable, you may define the $guarded property as an empty array:

```
class User extends Model
{
        /**
        * The attributes that aren't mass assignable.
        *
```

```
        * @var array
        */
        protected $guarded = [];
}
```

But you can also create a record without even changing `fillable` attribute in your model by using `forceCreate` method rather than `create`method

```
User::forceCreate([ 'first_name'=>
        'John', 'last_name' => 'Doe',
        'email'                 => 'john.doe@example.com',
        'password'          => bcrypt('changeme'),
]);
```

The following is an example of updating an existing `User` model by first loading it (by using `find`), modifying it, and then saving it:

```
$user = User::find(1);
$user->password = bcrypt('my_new_password');
$user->save();
```

To accomplish the same feat with a single function call, you may use the `update` method:

```
$user->update([
        'password' => bcrypt('my_new_password'),
]);
```

The `create` and `update`methods make working with large sets of data much simpler than having to set each key/value pair individually, as shown in the following examples:

> Note the use of `only` and `except`when gathering request data. It's important you specify the exact keys you want to allow/disallow to be updated, otherwise it's possible for an attacker to send additional fields with their request and cause unintended updates.

```
// Updating a user from specific request data
$data = Request::only(['first_name', 'email']);
$user->find(1);
$user->update($data);

// Create a user from specific request data
$data = Request::except(['_token', 'profile_picture', 'profile_name']);
```

Deleting

You can delete data after writing it to the database. You can either delete a model instance if you have retrieved one, or specify conditions for which records to delete.

To delete a model instance, retrieve it and call the `delete()`method:

```
$user = User::find(1);
$user->delete();
```

Alternatively, you can specify a primary key (or an array of primary keys) of the records you wish to delete via the `destroy()` method:

```
User::destroy(1); User::destroy([1, 2,
3]);
```

You can also combine querying with deleting:

```
User::where('age', '<', 21)->delete();
```

This will delete all users who match the condition.

> Note: When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be fired for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

Some times you don't want to permanently delete a record, but keep it around for auditing or reporting purposes. For this, Eloquent provides *soft deleting* functionality.

To add soft deletes functionality to your model, you need to import the `SoftDeletes` trait and add it to your Eloquent model class:

```
namespace Illuminate\Database\Eloquent\Model; namespace
Illuminate\Database\Eloquent\SoftDeletes;

class User extends Model
{
        use SoftDeletes;
```

When deleting a model, it will set a timestamp on a `deleted_at` timestamp column in the table for your model, so be sure to create the `deleted_at` column in your table first. Or in migration you should call `softDeletes()` method on your blueprint to add the `deleted_at`timestamp. Example:

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

Any queries will omit soft-deleted records. You can force-show them if you wish by using the `withTrashed()` scope:

```
User::withTrashed()->get();
```

If you wish to allow users to *restore* a record after soft-deleting (i.e. in a trash can-type area) then

you can use the `restore()` method:

```
$user = User::find(1);
$user->delete();
$user->restore();
```

To forcefully delete a record use the `forceDelete()` method which will truly remove the record from the database:

```
$user = User::find(1);
$user->forceDelete();
```

Change primary key and timestamps

By default, Eloquent models expect for the primary key to be named `'id'`. If that is not your case, you can change the name of your primary key by specifying the `$primaryKey` property.

```
class Citizen extends Model
{
	protected $primaryKey = 'socialSecurityNo';

	// ...
}
```

Now, any Eloquent methods that use your primary key (e.g. `find` or `findOrFail`) will use this new name.

Additionally, Eloquent expects the primary key to be an auto-incrementing integer. If your primary key is not an auto-incrementing integer (e.g. a GUID), you need to tell Eloquent by updating the `$incrementing` property to `false`:

```
class Citizen extends Model
{
	protected $primaryKey = 'socialSecurityNo'; public

	$incrementing = false;

	// ...
```

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to false:

```
class Citizen extends Model
{
	public $timestamps = false;

	// ...
}
```

If you need to customize the names of the columns used to store the timestamps, you may set the `CREATED_AT` and `UPDATED_AT`constants in your model:

```
class Citizen extends Model
{
        const CREATED_AT = 'date_of_creation'; const UPDATED_AT
        = 'date_of_last_update';

        // ...
```

Throw 404 if entity not found

If you want to automatically throw an exception when searching for a record that isn't found on a modal, you can use either

```
Vehicle::findOrFail(1);
```

or

```
Vehicle::where('make', 'ford')->firstOrFail();
```

If a record with the primary key of `1` is not found, a `ModelNotFoundException`is thrown. Which is essentially the same as writing (view source):

```
$vehicle = Vehicle::find($id);

if (!$vehicle) {
        abort(404);
}
```

Cloning Models

You may find yourself needing to clone a row, maybe change a few attributes but you need an efficient way to keep things DRY. Laravel provides a sort of 'hidden' method to allow you to do this functionality. Though it is completely undocumented, you need to search through the API to find it.

Using `$model->replicate()` you can easily clone a record

```
$robot = Robot::find(1);
$cloneRobot = $robot->replicate();
// You can add custom attributes here, for example he may want to evolve with an extra arm!
$cloneRobot->arms += 1;
$cloneRobot->save();
```

The above would find a robot that has an ID of 1, then clones it.

Read Eloquent online: https://riptutorial.com/laravel/topic/865/eloquent

Chapter 22: Eloquent : Relationship

Examples

Querying on relationships

Eloquent also lets you query on defined relationships, as show below:

```
User::whereHas('articles', function (Builder $query) {
        $query->where('published', '!=', true);
})->get();
```

This requires that your relationship method name is `articles` in this case. The argument passed into the closure is the Query Builder for the related model, so you can use any queries here that you can elsewhere.

Eager Loading

Suppose User model has a relationship with Article model and you want to eager load the related articles. This means the articles of the user will be loaded while retrieving user.

`articles` is the relationship name (method) in User model.

```
User::with('articles')->get();
```

if you have multiple relationship. for example articles and posts.

```
User::with('articles','posts')->get();
```

and to select nested relationships

```
User::with('posts.comments')->get();
```

Call more than one nested relationship

```
User::with('posts.comments.likes')->get()
```

Inserting Related Models

Suppose you have a `Post` model with a `hasMany`relationship with `Comment`. You may insert a `Comment` object related to a post by doing the following:

```
$post = Post::find(1);

$commentToAdd = new Comment(['message' => 'This is a comment.']);
```

```
$post->comments()->save($commentToAdd);
```

You can save multiple models at once using the `saveMany` function:

```
$post = Post::find(1);

$post->comments()->saveMany([
        new Comment(['message' => 'This a new comment']), new
        Comment(['message' => 'Me too!']),
        new Comment(['message' => 'Eloquent is awesome!'])
]);
```

Alternatively, there's also a `create` method which accepts a plain PHP array instead of an Eloquent model instance.

```
$post = Post::find(1);

$post->comments()->create([
        'message' => 'This is a new comment message'
]);
```

Introduction

Eloquent relationships are defined as functions on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful query builders, defining relationships as functions provides powerful method chaining and querying capabilities. For example, we may chain additional constraints on this posts relationship:

```
$user->posts()->where('active', 1)->get();
```

Navigate to parent topic

Relationship Types

One to Many

Lets say that each Post may have one or many comments and each comment belongs to just a single Post.

so the comments table will be having `post_id`. In this case the relationships will be as follows.

Post Model

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

If the foreign key is other than `post_id`, for example the foreign key is `example_post_id`.

```
public function comments()
{
    return $this->belongsTo(Post::class, 'example_post_id');
}
```

and plus, if the local key is other than `id`, for example the local key is `other_id`

```
public function comments()
{
    return $this->belongsTo(Post::class, 'example_post_id', 'other_id');
}
```

Comment Model

defining inverse of one to many

```
public function post()
{
    return $this->hasMany(Comment::class);
}
```

One to One

How to associate between two models (example: `User` and `Phone` model)

App\User

```
<?php namespace

App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
        /**
          * Get the phone record associated with the user.
          */
        public function phone()
        {
                return $this->hasOne('Phone::class', 'foreign_key', 'local_key');
```

App\Phone

```
<?php namespace

App;

use Illuminate\Database\Eloquent\Model;
```

```
class Phone extends Model
{
	/**
	  * Get the user that owns the phone.
	  */
	public function user()
	{
		return $this->belongsTo('User::class', 'foreign_key', 'local_key');
	}
```

foreign_key : By default Eloquent will assume this value to be other_model_name_id(in this case user_id and phone_id), change it if it isn't the case.

local_key : By default Eloquent will assume this value to be id(current model primary key), change it if it isn't the case.

> If your database filed name as per laravel standard, you don't need to provide foreign key and local key in relationship declaration

Explanation

Many to Many

Lets say there is roles and permissions. Each role may belongs to many permissions and each permission may belongs to many role. so there will be 3 tables. two models and one pivot table. a roles, users and permission_roletable.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class);
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class);
}
```

Note: 1

consider following while using different table name for pivot table.

Suppose if you want to use role_permission instead of permission_role, as eloquent uses alphabetic order for building the pivot key names. you will need to pass pivot table name as second parameter as follows.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class, 'role_permission');
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class, 'role_permission');
}
```

Note: 2

consider following while using different key names in pivot table.

Eloquent assumes that if no keys are passed as third and fourth parameters that it will be the singular table names with `_id`. so it assumes that the pivot will be having `role_id` and `permission_id` fields. If keys other than these are to be used it should be passed as third and fourth parameters.

Lets say if `other_role_id`instead of `role_id`and `other_permission_id`instead of `permission_id`is to be used. So it would be as follows.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class, 'role_permission', 'other_role_id', 'other_permission_id');
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class, 'role_permission', 'other_permission_id', 'other_role_id');
}
```

Polymorphic

Polymorphic relations allow a model to belong to more than one other model on a single association. A good example would be images, both a user and a product can have an image. The table structure might look as follows:

```
user
    id - integer name -
    string email - string

product
```

```
        id - integer title -
        string SKU - string

image
        id - integer url -
        string
        imageable_id - integer
        imageable_type - string
```

The important columns to look at are in the images table. The `imageable_id` column will contain the ID value of the user or product, while the `imageable_type` column will contain the class name of the owning model. In your models, you setup the relations as follows:

```
<?php namespace

App;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
        /**
          * Get all of the owning imageable models.
          */
        public function imageable()
        {
                return $this->morphTo();
        }
}

class User extends Model
{
        /**
          * Get all of the user's images.
          */
        public function images()
        {
                return $this->morphMany('Image::class', 'imageable');
        }
}

class Product extends Model
{
        /**
          * Get all of the product's images.
          */
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In our case, that is the `imageable` method on the Image model. So, we will access that method as a dynamic property

```
$image = App\Image::find(1);

$imageable = $image->imageable;
```

This `imageable` will return either a User or a Product.

Many To Many

Lets say there is roles and permissions. Each role may belongs to many permissions and each permission may belongs to many role. so there will be 3 tables. two models and one pivot table. a `roles`, `users` and `permission_role`table.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class);
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class);
}
```

Note: 1

consider following while using different table name for pivot table.

Suppose if you want to use `role_permission` instead of `permission_role`, as eloquent uses alphabetic order for building the pivot key names. you will need to pass pivot table name as second parameter as follows.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class, 'role_permission');
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class, 'role_permission');
}
```

Note: 2

consider following while using different key names in pivot table.

Eloquent assumes that if no keys are passed as third and fourth parameters that it will be the singular table names with `_id`. so it assumes that the pivot will be having `role_id` and `permission_id` fields. If keys other than these are to be used it should be passed as third and fourth parameters.

Lets say if `other_role_id`instead of `role_id`and `other_permission_id`instead of `permission_id`is to be used. So it would be as follows.

Role Model

```
public function permissions()
{
    return $this->belongsToMany(Permission::class, 'role_permission', 'other_role_id', 'other_permission_id');
}
```

Permission Model

```
public function roles()
{
    return $this->belongsToMany(Roles::class, 'role_permission', 'other_permission_id', 'other_role_id');
}
```

Accessing Intermediate table using withPivot()

Suppose you have a third column '**permission_assigned_date**' in the pivot table . By default, only the model keys will be present on the pivot object. Now to get this column in query result you need to add the name in withPivot() function.

```
    public function permissions()
            {
                return $this->belongsToMany(Permission::class, 'role_permission', 'other_role_id', 'other_permission_id')-
>withPivot('permission_assigned_date');
            }
```

Attaching / Detaching

Eloquent also provides a few additional helper methods to make working with related models more convenient. For example, let's imagine a user can have many roles and a role can have many permissions. To attach a role to a permission by inserting a record in the intermediate table that joins the models, use the attach method:

```
$role= App\Role::find(1);
$role->permissions()->attach($permissionId);
```

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

```
$rol->roles()->attach($permissionId, ['permission_assigned_date' => $date]);
```

Similarly, To remove a specific permission against a role use detach function

```
$role= App\Role::find(1);
//will remove permission 1,2,3 against role 1
$role->permissions()->detach([1, 2, 3]);
```

Syncing Associations

You may also use the sync method to construct many-to-many associations. The sync method accepts an array of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the given array will exist in the intermediate table:

```
//will keep permission id's 1,2,3 against Role id 1

$role= App\Role::find(1)
$role->permissions()->sync([1, 2, 3]);
```

Read Eloquent : Relationship online: https://riptutorial.com/laravel/topic/7960/eloquent---relationship

Chapter 23: Eloquent: Accessors & Mutators

Introduction

Accessors and mutators allow you to format Eloquent attribute values when you retrieve or set them on model instances. For example, you may want to use the Laravel encrypter to encrypt a value while it is stored in the database, and then automatically decrypt the attribute when you access it on an Eloquent model. In addition to custom accessors and mutators, Eloquent can also automatically cast date fields to Carbon instances or even cast text fields to JSON.

Syntax

- set{ATTRIBUTE}Attribute($attribute) // in camel case

Examples

Defining An Accessors

```
<?php namespace

App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
        /**
          *  Get the user's first name.
          *
          *  @param     string         $value
          *  @return string
          */
        public function getFirstNameAttribute($value)
        {
                return ucfirst($value);
```

Getting Accessor:

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may simply access the `first_name` attribute on a model instance:

```
$user = App\User::find(1);
$firstName = $user->first_name;
```

Defining a Mutator

```
class User extends Model
{
	public function setPasswordAttribute($password)
	{
		$this->attributes['password'] = bcrypt($password);
	}
	...
```

Above code does "bcrypting" each time password property is set.

```
$user = $users->first();
$user->password = 'white rabbit'; //laravel calls mutator on background
$user->save(); // password is bcrypted and one does not need to call bcrypt('white rabbit')
```

Read Eloquent: Accessors & Mutators online: https://riptutorial.com/laravel/topic/8305/eloquent--accessors---mutators

Chapter 24: Eloquent: Model

Examples

Making a Model

Model creation

Model classes must extend `Illuminate\Database\Eloquent\Model`. The default location for models is the `/app` directory.

A model class can be easily generated by the Artisan command:

```
php artisan make:model [ModelName]
```

This will create a new PHP file in `app/` by default, which is named `[ModelName].php`, and will contain all the boilerplate for your new model, which includes the class, namespace, and using's required for a basic setup.

If you want to create a migration file along with your Model, use the following command, where `-m` will also generate the migration file:

```
php artisan make:model [ModelName] -m
```

In addition to creating the model, this creates a database migration that is hooked up to the model. The database migration PHP file is located by default in `database/migrations/`. This does not--by default--include anything other than the id and `created_at`/`updated_at` columns, so you will need to edit the file to provide additional columns.

Note that you will have to run the migration (once you have set up the migration file) in order for the model to start working by using `php artisan migrate` from project root

In addition, if you wish to add a migration later, after making the model, you can do so by running:

```
php artisan make:migration [migration name]
```

Say for example you wanted to create a model for your Cats, you would have two choices, to create with or without a migration. You would chose to create without migration if you already had a cats table or did not want to create one at this time.

For this example we want to create a migration because we don't already have a table so would run the following command.

```
php artisan make:model Cat -m
```

This command will create two files:

1. In the App folder: `app/Cat.php`
2. In the database folder: `database/migrations/timestamp_creat_cats_table.php`

The file we are interested in is the latter as it is this file that we can decide what we want the table to look like and include. For any predefined migration we are given an auto incrementing id column and a timestamps columns.

The below example of an extract of the migration file includes the above predefined columns as well as the addition of a the name of the cat, age and colour:

```
public function up()
    {
        Schema::create('cats', function (Blueprint $table) {

            $table->increments('id');           //Predefined ID
            $table->string('name');             //Name
            $table->integer('age');             //Age
            $table->string('colour');           //Colour


        });
    }
```

So as you can see it is relatively easy to create the model and migration for a table. Then to execute the migration and create it in your data base you would run the following command:

```
php artisan migrate
```

Which will migrate any outstanding migrations to your database.

Model File Location

Models can be stored anywhere thanks to PSR4.

By default models are created in the `app` directory with the namespace of `App`. For more complex applications it's usually recommended to store models within their own folders in a structure that makes sense to your apps architecture.

For example, if you had an application that used a series of fruits as models, you could create a folder called `app/Fruits` and within this folder you create `Banana.php` (keeping the StudlyCase naming convention), you could then create the Banana class in the `App\Fruits` namespace:

```
namespace App\Fruits;

use Illuminate\Database\Eloquent\Model; class Banana

extends Model {
    // Implementation of "Banana" omitted
}
```

Model configuration

Eloquent follows a "convention over configuration" approach. By extending the base `Model` class, all models inherit the properties listed below. Unless overridden, the following default values apply:

Property	Description	Default
protected $connection	DB connection name	Default DB connection
protected $table	Table name	By default, the class name is converted to `snake_case` and pluralized. For example, `SpecialPerson` becomes special_people
protected $primaryKey	Table PK	id
public $incrementing	Indicates if the IDs are auto-incrementing	true
public $timestamps	Indicates if the model should be timestamped	true
const CREATED_AT	Name of the creation timestamp column	created_at
const UPDATED_AT	Name of the modification timestamp column	updated_at
protected $dates	Attributes that should be mutated to DateTime, in addition to the timestamps attributes	[]
protected $dateFormat	Format in which date attributes will be persisted	Default for current SQL dialect.
protected $with	Relationships to eagerload with model	[]
protected $hidden	Attributes omitted in model serialization	[]
protected $visible	Attributes allowed in model serialization	[]
protected $appends	Attribute accessors added to model serialization	[]

protected	Attributes that are mass-	[]

Property	Description	Default
$fillable	assignable	
protected $guarded	Attributes that are black-listed from mass assignment	[*] (All attributes)
protected $touches	The relationships that should be touched on save	[]
protected $perPage	The number of models to return for pagination.	15

5.0

Property	Description	Default
protected $casts	Attributes that should be casted to native types	[]

Update an existing model

```
$user = User::find(1);
$user->name = 'abc';
$user->save();
```

You can also update multiple attributes at once using `update`, which does not require using `save` afterwards:

```
$user = User::find(1);
$user->update(['name' => 'abc', 'location' => 'xyz']);
```

You can also update a model(s) without querying it beforehand:

```
User::where('id', '>', 2)->update(['location' => 'xyz']);
```

If you don't want to trigger a change to the `updated_at` timestamp on the model then you can pass the `touch` option:

```
$user = User::find(1);
$user->update(['name' => 'abc', 'location' => 'xyz'], ['touch' => false]);
```

Read Eloquent: Model online: https://riptutorial.com/laravel/topic/7984/eloquent--model

Chapter 25: Error Handling

Remarks

Remember to set up your application for emailing by ensuring proper configuration of config/mail.php

Also check to make sure ENV variables are properly set.

This example is a guide and is minimal. Explore, modify and style the view as you wish. Tweak the code to meet your needs. For example, set the recepient in your *.env* file

Examples

Send Error report email

Exceptions in Laravel are handled by *App\Exceptions\Handler.php*

This file contains two functions by default. Report & Render. We will only be using the first

```
public function report(Exception $e)
```

> The report method is used to log exceptions or send them to an external service like BugSnag. By default, the report method simply passes the exception to the base class where the exception is logged. However, you are free to log exceptions however you wish.

Essentially this function just forwards the error and does nothing. Therefore, we can insert business logic to perform operations based on the error. For this example we will be sending an email containing the error information.

```
public function report(Exception $e)
{
    if ($e instanceof \Exception) {
        // Fetch the error information we would like to
        // send to the view for emailing
        $error['file']    = $e->getFile();
        $error['code']    = $e->getCode();
        $error['line']    = $e->getLine();
        $error['message'] = $e->getMessage();
        $error['trace']   = $e->getTrace();

        // Only send email reports on production server if(ENV('APP_ENV') ==
        "production"){
            #1. Queue email for sending on "exceptions_emails" queue #2. Use the
            emails.exception_notif view shown below
            #3. Pass the error array to the view as variable $e Mail::queueOn('exception_emails', 'emails.exception_notif',
```

```
                $m->from(ENV("MAIL_FROM"), ENV("MAIL_NAME"));
                $m->to("webmaster@laravelapp.com", "Webmaster");
            });

        }
    }

    // Pass the error on to continue processing return
    parent::report($e);
```

The view for the email ("emails.exception_notif") is below

```
<?php
$action = (\Route::getCurrentRoute()) ? \Route::getCurrentRoute()->getActionName() : "n/a";
$path = (\Route::getCurrentRoute()) ? \Route::getCurrentRoute()->getPath() : "n/a";
$user = (\Auth::check()) ? \Auth::user()->name : 'no login';
?>

There was an error in your Laravel App<br />

<hr />
<table border="1" width="100%">
    <tr><th >User:</th><td>{{ $user }}</td></tr>
    <tr><th >Message:</th><td>{{ $e['message'] }}</td></tr>
    <tr><th >Action:</th><td>{{ $action }}</td></tr>
    <tr><th >URI:</th><td>{{ $path }}</td></tr>
    <tr><th >Line:</th><td>{{ $e['line'] }}</td></tr>
```

Catching application wide ModelNotFoundException

app\Exceptions\Handler.php

```
public function render($request, Exception $exception)
{
    if ($exception instanceof ModelNotFoundException) { abort(404);
    }

    return parent::render($request, $exception);
}
```

You can catch / handle any exception that is thrown in Laravel.

Read Error Handling online: https://riptutorial.com/laravel/topic/2858/error-handling

Chapter 26: Events and Listeners

Examples

Using Event and Listeners for sending emails to a new registered user

Laravel's events allows to implement the Observer pattern. This can be used to send a welcome email to a user whenever they register on your application.

New events and listeners can be generated using the artisan command line utility after registering the event and their particular listener in `App\Providers\EventServiceProvider` class.

```
protected $listen = [ 'App\Events\NewUserRegistered' =>
        [
                'App\Listeners\SendWelcomeEmail',
        ],
];
```

Alternate notation:

```
protected $listen = [
        \App\Events\NewUserRegistered::class => [
                \App\Listeners\SendWelcomeEmail::class,
        ],
];
```

Now execute `php artisan generate:event`. This command will generate all the corresponding events and listeners mentioned above in `App\Events` and `App\Listeners`directories respectively.

We can have multiple listeners to a single event like

```
protected $listen = [ 'Event' =>
        [
                'Listner1', 'Listener2'
        ],
];
```

`NewUserRegistered` is just a wrapper class for the newly registered User model:

```
class NewUserRegistered extends Event
{
        use SerializesModels; public

        $user;

        /**
          *  Create a new event instance.
          *
          *  @return void
          */
```

```
        {
                $this->user = $user;
        }
}
```

This `Event` will be handled by the `SendWelcomeEmail`listener:

```
class SendWelcomeEmail
{
        /**
          *  Handle the event.
          *
          *  @param     NewUserRegistered            $event
          */
        public function handle(NewUserRegistered $event)
        {
                //send the welcome email to the user
                $user = $event->user;
                Mail::send('emails.welcome', ['user' => $user], function ($message) use ($user) {
                        $message->from('hi@yourdomain.com', 'John Doe');
                        $message->subject('Welcome aboard '.$user->name.'!');
                        $message->to($user->email);
                });
```

The last step is to call/fire the event whenever a new user registers. This can be done in the controller, command or service, wherever you implement the user registration logic:

```
event(new NewUserRegistered($user));
```

Read Events and Listeners online: https://riptutorial.com/laravel/topic/4687/events-and-listeners

Chapter 27: Filesystem / Cloud Storage

Examples

Configuration

The filesystem configuration file is located at `config/filesystems.php`. Within this file you may configure all of your "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver is included in the configuration file. So, simply modify the configuration to reflect your storage preferences and credentials!

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

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

Of course, you may configure as many disks as you like, and may even have multiple disks that use the same driver.

When using the local driver, note that all file operations are relative to the root directory defined in your configuration file. By default, this value is set to the `storage/app directory`. Therefore, the following method would store a file in `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Basic Usage

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

Retrieving A Particular Disk

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

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

Determining If A File Exists

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

Calling Methods On The Default Disk

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

```
        //
}
```

Retrieving A File's Contents

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

Setting A File's Contents

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

Prepend To A File

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

Append To A File

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

Delete A File

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

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

Copy A File To A New Location

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

Move A File To A New Location

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

Get File Size

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

Get The Last Modification Time (UNIX)

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

Get All Files Within A Directory

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

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

Get All Directories Within A Directory

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

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

Create A Directory

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

Delete A Directory

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

Custom Filesystems

Laravel's Flysystem integration provides drivers for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application. Don't worry, it's not too hard!

In order to set up the custom filesystem you will need to create a service provider such as `DropboxFilesystemServiceProvider`. In the provider's `boot` method, you can inject an instance of the `Illuminate\Contracts\Filesystem\Factory` contract and call the `extend` method of the injected instance. Alternatively, you may use the `Disk` facade's `extend`method.

The first argument of the `extend` method is the name of the driver and the second is a Closure that receives the `$app` and `$config`variables. The resolver Closure must return an instance of `League\Flysystem\Filesystem`.

> Note: The $config variable will already contain the values defined in `config/filesystems.php` for the specified disk. Dropbox Example

```
<?php namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Dropbox\Client as DropboxClient;
use League\Flysystem\Dropbox\DropboxAdapter; use
Illuminate\Support\ServiceProvider;

class DropboxFilesystemServiceProvider extends ServiceProvider { public function boot()
    {
        Storage::extend('dropbox', function($app, $config)
        {
            $client = new DropboxClient($config['accessToken'], $config['clientIdentifier']);

            return new Filesystem(new DropboxAdapter($client));
        });
```

```
    }

    public function register()
    {
        //
    }

```

Creating symbolic link in a web server using SSH

In Laravel documentation, a symbolic link (symlink or soft link) from public/storage to storage/app/public should be created to make files accessible from the web.

(THIS PROCEDURE WILL CREATE SYMBOLIC LINK WITHIN THE LARAVEL PROJECT DIRECTORY)

Here are the steps on how you can create symbolic link in your Linux web server using SSH client:

1. Connect and login to your web server using SSH client (e.g. PUTTY).
2. Link **storage/app/public** to **public/storage** using the syntax

   ```
   ln -s target_path link_path
   ```

   Example (in CPanel File Directory)

   ```
   ln -s /home/cpanel_username/project_name/storage/app/public
   /home/cpanel_sername/project_name/public/storage
   ```

*(A folder named **storage** will be created to link path with an indicator **>>>** on the folder icon.)*

Read Filesystem / Cloud Storage online: https://riptutorial.com/laravel/topic/3040/filesystem---cloud-storage

Chapter 28: Form Request(s)

Introduction

Custom requests (or Form Requests) are useful in situations when one wants to **authorize** & **validate** a request before hitting the controller method.

One may think of two practical uses, **creating** & **updating** a record while each action has a different set of validation (or authorization) rules.

Using Form Requests is trivial, one has to type-hint the request class in method.

Syntax

- php artisan make:request name_of_request

Remarks

Requests are useful when separating your validation from Controller. It also allows you to check if the request is authorized.

Examples

Creating Requests

```
php artisan make:request StoreUserRequest

php artisan make:request UpdateUserRequest
```

> **Note**: You can also consider using names like **StoreUser** or **UpdateUser** (without **Request** appendix) since your FormRequests are placed in folder `app/Http/Requests/`.

Using Form Request

Lets say continue with User example (you may have controller with store method and update method). To use FormRequests you use type-hinting the specific request.

```
...

public function store(App\Http\Requests\StoreRequest $request, App\User $user) {
        //by type-hinting the request class, Laravel "runs" StoreRequest
        //before actual method store is hit

        //logic that handles storing new user
        //(both email and password has to be in $fillable property of User model
```

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

...

public function update(App\Http\Requests\UpdateRequest $request, App\User $users, $id) {
        //by type-hinting the request class, Laravel "runs" UpdateRequest
        //before actual method update is hit

        //logic that handles updating a user
        //(both email and password has to be in $fillable property of User model
        $user = $users->findOrFail($id);
        $user->update($request->only(['password'])); return redirect()-
```

Handling Redirects after Validation

Sometimes you may want to have some login to determine where the user gets redirected to after submitting a form. Form Requests give a variety of ways.

By default there are 3 variables declared in the Request `$redirect`, `$redirectRoute` and `$redirectAction`.

On top of those 3 variables you can override the main redirect handler `getRedirectUrl()`. A

sample request is given below explaining what you can do.

```
<?php namespace App;

use Illuminate\Foundation\Http\FormRequest as Request; class SampleRequest

extends Request {

        // Redirect to the given url public
        $redirect;

        // Redirect to a given route public
        $redirectRoute;

        // Redirect to a given action public
        $redirectAction;


        /**
          *   Get the URL to redirect to on a validation error.
          *
          *   @return string
          */
        protected function getRedirectUrl()
        {

                // If no path is given for `url()` it will return a new instance of
```

```
        //`url()` provides several methods you can chain such as

        // Get the current URL return url()-
        >current();

        // Get the full URL of the current request return url()->full();

        // Go back
        return url()->previous();

        // Or just redirect back return
        redirect()->back();
}


/**
  *  Get the validation rules that apply to the request.
  *
  *  @return array
  */
public function rules()
{
       return [];
}

/**
  *  Determine if the user is authorized to make this request.
  *
  *  @return bool
  */
public function authorize()
```

Read Form Request(s) online: https://riptutorial.com/laravel/topic/6329/form-request-s-

Chapter 29: Getting started with laravel-5.3

Remarks

This section provides an overview of what laravel-5.3 is, and why a developer might want to use it.

It should also mention any large subjects within laravel-5.3, and link out to the related topics. Since the Documentation for laravel-5.3 is new, you may need to create initial versions of those related topics.

Examples

Installing Laravel

Requirements:
You need PHP >= 5.6.4 and Composerinstalled on your machine. You can check version of both by using command:
For PHP:

```
php -v
```

Output like this:

```
PHP 7.0.9 (cli) (built: Aug 26 2016 06:17:04) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```

For Composer
You can run command on your terminal/CMD:

```
composer --version
```

Output like this:

```
composer version 1.2.1 2016-09-12 11:27:19
```

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

Via Laravel Installer

First, download the Laravel installer using Composer:

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

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

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

```
laravel new blog
```

Via Composer Create-Project

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

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

Setup

After you are complete with the Laravel installation, you will need to set `permissions` for the storage and Bootstrap folders.

> **Note:** Setting `permissions` is one of the most important processes to complete while installing Laravel.

Local Development Server

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

```
php artisan serve
```

Open your browser request url `http://localhost:8000`

Server Requirements

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

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

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

- Tokenizer PHP Extension
- XML PHP Extension

Local Development Server

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

```
php artisan serve
```

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

Also it's possible to use a custom port, something like `8080`. You can do this with the `--port` option.

```
php artisan serve --port=8080
```

If you have a local domain in your hosts file, you can set the hostname. This can be done by the `--host` option.

```
php artisan serve --host=example.dev
```

You can also run on a custom host and port, this can be done by the following command.

```
php artisan serve --host=example.dev --port=8080
```

Hello World Example (Basic) and with using a view

The basic example
Open `routes/web.php` file and paste the following code in file:

```
Route::get('helloworld', function () { return '<h1>Hello
      World</h1>';
});
```

here '**helloworld**' will act as page name you want to access,

and if you don't want to create blade file and still want to access the page directly then you can use laravel routing this way

now type `localhost/helloworld` in browser address bar and you can access page displaying Hello World.

The next step.
So you've learned how to create a very simple Hello World! page by returning a hello world sentence. But we can make it a bit nicer!

Step 1.

We'll start again at our routes/web.php file now instead of using the code above we'll use the following code:

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

The return value this time is not just a simple helloworld text but a view. A view in Laravel is simply a new file. This file "helloworld" contains the HTML and maybe later on even some PHP of the Helloworld text.

Step 2.
Now that we've adjusted our route to call on a view we are going to make the view. Laravel works with blade.php files in views. So, in this case, our route is called helloworld. So our view will be called helloworld.blade.php

We will be creating the new file in the resources/views directory and we will call it helloworld.blade.php

Now we'll open this new file and edit it by creating our Hello World sentence. We can add multiple different ways to get our sentence as in the example below.

```
<html>
    <body>
        <h1> Hello World! </h1>

        <?php
            echo "Hello PHP World!";
        ?>

    </body>
```

now go to your browser and type your route again like in the basic example: localhost/helloworld
you'll see your new created view with all of the contents!

Hello World Example (Basic)

Open routes file. Paste the following code in:

```
Route::get('helloworld', function () { return '<h1>Hello
    World</h1>';
});
```

after going to route http://localhost/helloworld it displays Hello World. The

routes file is located /routes/web.php

Web Server Configuration for Pretty URLs

If you installed Laravel via Composer or the Laravel installer, below configuration you will need.

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

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

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
```

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

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

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

Read Getting started with laravel-5.3 online: https://riptutorial.com/laravel/topic/8602/getting-started-with-laravel-5-3

Chapter 30: Helpers

Introduction

Laravel helpers are the globally accessible functions defined by the framework. It can be directly called and independently used anywhere within the application without needing to instantiating an object or importing class.

There are helpers for manipulating *Arrays*, *Paths*, *Strings*, *URLs*, etc

Examples

Array methods

array_add()

This method is used to add new key value pairs to an array.

```
$array = ['username' => 'testuser'];

$array = array_add($array, 'age', 18);
```

result

```
['username' => 'testuser', 'age' => 18]
```

String methods

camel_case()

This method changes a string to camel case

```
camel_case('hello_world');
```

result

```
HelloWorld
```

Path mehods

Path methods helps easy access to application related paths easily from anywhere.

public_path()

This method returns the fully qualified public path of the application. which is the public directory.

```
$path = public_path();
```

Urls

url()

The url function generates a fully qualified URL to the given path.

if your site is hello.com

```
echo url('my/dashboard');
```

would return

```
hello.com/my/dashboard
```

if nothing is passed to the url method it would return an instance of Illuminate\Routing\UrlGenerator, and it could be used like this would

return current url

```
echo url()->current();
```

would return full url

```
echo url()->full();
```

would return previous url

```
echo url()->previous();
```

Read Helpers online: https://riptutorial.com/laravel/topic/8827/helpers

Chapter 31: HTML and Form Builder

Examples

Installation

HTML and Form Builder is not a core component since Laravel 5, so we need to install it separately:

```
composer require laravelcollective/html "~5.0"
```

Finally in `config/app.php` we need to register the service provider, and the facades aliases like this:

```
'providers' => [
        // ... Collective\Html\HtmlServiceProvider::class,
        // ...
],

'aliases' => [
      // ...
    'Form' => Collective\Html\FormFacade::class, 'Html' =>
    Collective\Html\HtmlFacade::class,
      // ...
]
```

Full docs are available on Forms & HTML

Read HTML and Form Builder online: https://riptutorial.com/laravel/topic/3672/html-and-form-builder

Chapter 32: Installation

Examples

Installation

Laravel applications are installed and managed with Composer, a popular PHP dependency manager. There are two ways to create a new Laravel application.

Via Composer

```
$ composer create-project laravel/laravel [foldername]
```

Or

```
$ composer create-project --prefer-dist laravel/laravel [foldername]
```

Replace **[foldername]** with the name of the directory you want your new Laravel application installed to. It must not exist before installation. You may also need to add the Composer executable to your system path.

If want to create a Laravel project using a specific version of the framework, you can provide a version pattern, otherwise your project will use the latest available version.

If you wanted to create a project in Laravel 5.2 for example, you'd run:

```
$ composer create-project --prefer-dist laravel/laravel 5.2.*
```

Why --prefer-dist

There are two ways of downloading a package: `source` and `dist`. For stable versions Composer will use the `dist` by default. The `source` is a version control repository. If `--prefer-source` is enabled, Composer will install from source if there is one.

`--prefer-dist`is the opposite of `--prefer-source`, and tells Composer to install from `dist`if possible. This can speed up installs substantially on build servers and in other use cases where you typically do not run vendor updates. It also allows avoiding problems with Git if you do not have a proper setup.

Via the Laravel installer

Laravel provides a helpful command line utility to quickly create Laravel applications. First, install the installer:

```
$ composer global require laravel/installer
```

> You have to make sure that the Composer binaries folder is within your $PATH variable to execute the Laravel installer.
>
> First, look if it already is in your $PATH variable
>
> ```
> echo $PATH
> ```
>
> If everything is correct, the output should contain something like this:
>
> ```
> Users/yourusername/.composer/vendor/bin
> ```
>
> If not, edit your `.bashrc` or, if your using ZSH, your `.zshrc` so it contains the path to your Composer vendor directory.

Once installed, this command will create a fresh Laravel installation in the directory you specify.

```
laravel new [foldername]
```

You can also use `.` (a dot) in place of **[foldername]** to create the project in the current working directory without making a sub-directory.

Running the application

Laravel comes bundled with a PHP-based web server which can be started by running

```
$ php artisan serve
```

By default, the HTTP server will use port 8000, but if the port is already in use or if you want to run multiple Laravel applications at once, you can use the `--port` flag to specify a different port:

```
$ php artisan serve --port=8080
```

The HTTP server will use `localhost` as the default domain for running the application, but you can use the `--host` flag to specify a different address:

```
$ php artisan serve --host=192.168.0.100 --port=8080
```

Using a different server

If you prefer to use a different web server software, some configuration files are provided for you inside the `public` directory of your project; `.htaccess` for Apache and `web.config` for ASP.NET. For other software such as NGINX, you can convert the Apache configurations using various online tools.

The framework needs the web server user to have write permissions on the following directories:

- /storage
- /bootstrap/cache

On *nix operating systems this can be achieved by

```
chown -R www-data:www-data storage bootstrap/cache chmod -R
ug+rwx storage bootstrap/cache
```

(where `www-data` is the name and group of the web server user)

The web server of your choice should be configured to serve content from your project's `/public` directory, which is usually done by setting it as the document root. The rest of your project should not be accessible through your web server.

If you set everything up properly, navigating to your website's URL should display the default landing page of Laravel.

Requirements

The Laravel framework has the following requirements:

5.3

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

5.1 (LTS)5.2

- PHP >= 5.5.9
- PDO PHP Extension
- Laravel 5.1 is the first version of Laravel to support PHP 7.0.

5.0

- PHP >= 5.4, PHP < 7
- OpenSSL PHP Extension
- Tokenizer PHP Extension
- Mbstring PHP Extension
- JSON PHP extension (only on PHP 5.5)

4.2

- PHP >= 5.4
- Mbstring PHP Extension
- JSON PHP extension (only on PHP 5.5)

Hello World Example (Using Controller and View)

1. Create a Laravel application:

```
$ composer create-project laravel/laravel hello-world
```

2. Navigate to the project folder, e.g.

```
$ cd C:\xampp\htdocs\hello-world
```

3. Create a controller:

```
$ php artisan make:controller HelloController --resource
```

This will create the file **app/Http/Controllers/HelloController.php**. The `--resource` option will generate CRUD methods for the controller, e.g. index, create, show, update.

4. Register a route to HelloController's `index` method. Add this line to **app/Http/routes.php** *(version 5.0 to 5.2)* or **routes/web.php** *(version 5.3)*:

```
Route::get('hello', 'HelloController@index');
```

To see your newly added routes, you can run `$ php artisan route:list`

5. Create a Blade template in the `views` directory:

resources/views/hello.blade.php:

```
<h1>Hello world!</h1>
```

6. Now we tell index method to display the **hello.blade.php** template:

app/Http/Controllers/HelloController.php

```
<?php

namespace App\Http\Controllers; use

Illuminate\Http\Request; use

App\Http\Requests;

class HelloController extends Controller
{
        /**
          *  Display a listing of the resource.
          *
          *  @return \Illuminate\Http\Response
          */
```

```
        return view('hello');
    }

    // ... other resources are listed below the index one above
```

You can serve your app using the following PHP Artisan Command: **php artisan serve**; it will show you the address at which you can access your application *(usually at http://localhost:8000 by default)*.

Alternatively, you may head over directly to the appropriate location in your browser; *in case you are using a server like XAMPP (either: http://localhost/hello-world/public/hello should you have installed your Laravel instance, **hello-world**, directly in your **xampp/htdocs** directory as in: having executed the step 1 of this Hello Word from your command line interface, pointing at your **xampp/htdocs** directory)*.

Hello World Example (Basic)

Open routes file. Paste the following code in:

```
Route::get('helloworld', function () { return '<h1>Hello
    World</h1>';
});
```

after going to route localhost/helloworld it displays **Hello World**. The

routes file is located:

5.3

For Web

```
routes/web.php
```

For APIs

```
routes/api.php
```

5.25.1 (LTS)5.0

```
app/Http/routes.php
```

4.2

```
app/routes.php
```

Installation using LaraDock (Laravel Homestead for Docker)

LaraDock is a Laravel Homestead like development environment but for Docker instead of Vagrant. https://github.com/LaraDock/laradock

Installation

*Requires Git and Docker

Clone the LaraDock repository:

A. If you already have a Laravel project, clone this repository on your Laravel root directory:

```
git submodule add https://github.com/LaraDock/laradock.git
```

B. If you don't have a Laravel project, and you want to install Laravel from Docker, clone this repo anywhere on your machine:

```
git clone https://github.com/LaraDock/laradock.git
```

Basic Usage

1. Run Containers: (Make sure you are in the laradock folder before running the docker-compose commands).

   Example: Running NGINX and MySQL: `docker-compose up -d nginx mysql`

   There are a list of available containers you can select to create your own combinations.

   `nginx`, `hhvm`, `php-fpm`, `mysql`, `redis`, `postgres`, `mariadb`, `neo4j`, `mongo`, `apache2`, `caddy`, `memcached`, `beanstalkd`, `beanstalkd-console`, `workspace`

2. Enter the Workspace container, to execute commands like (Artisan, Composer, PHPUnit, Gulp, ...).

   ```
   docker-compose exec workspace bash
   ```

3. If you don't have a Laravel project installed yet, follow the step to install Laravel from a Docker container.

   a. Enter the Workspace container.

   b. Install Laravel. `composer create-project laravel/laravel my-cool-app "5.3.*"`

4. Edit the Laravel configurations. Open your Laravel's .env file and set the DB_HOST to your mysql:

   ```
   DB_HOST=mysql
   ```

5. Open your browser and visit your localhost address.

Read Installation online: https://riptutorial.com/laravel/topic/7961/installation

Chapter 33: Installation Guide

Remarks

This section provides an overview of what laravel-5.4 is, and why a developer might want to use it.

It should also mention any large subjects within laravel-5.4, and link out to the related topics. Since the Documentation for laravel-5.4 is new, you may need to create initial versions of those related topics.

Examples

Installation

Detailed instructions on getting laravel set up or installed.

composer is required for installing laravel easily.

There are 3 methods of installing laravel in your system:

1. **Via Laravel Installer**

   Download the Laravel installer using composer

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

   Before using composer we need to add ~/.composer/vendor/bin to PATH. After installation has finished we can use laravel new command to create a new project in Laravel.

   Example:

   ```
   laravel new {folder name}
   ```

   This command creates a new directory named as site and a fresh Laravelinstallation with all other dependencies are installed in the directory.

2. **Via Composer Create-Project**

   You can use the command in the terminal to create a new Laravel app:

   ```
   composer create-project laravel/laravel {folder name}
   ```

3. **Via Download**

   Download Laravel and unzip it.

   1. composer install

2. Copy .env.exampleto .envvia teminalor manually.

```
cp .env.example .env
```

3. Open .env file and set your database, email, pusher, etc. (if needed)
4. php artisan migrate (if database is setup)
5. php artisan key:generate
6. php artisan serve
7. Go to localhost:8000 to view the site

Laravel docs

Hello World Example (Basic)

Accessing pages and outputting data is fairly easy in Laravel. All of the page routes are located in app/routes.php. There are usually a few examples to get you started, but we're going to create a new route. Open your app/routes.php, and paste in the following code:

```
Route::get('helloworld', function () { return '<h1>Hello
    World</h1>';
});
```

This tells Laravel that when someone accesses http://localhost/helloworld in a browser, it should run the function and return the string provided.

Hello World Example With Views and Controller

Assuming we have a working laravel application running in, say, "mylaravel.com",we want our application to show a "Hello World" message when we hit the URL http://mylaravel.com/helloworld . It involves the creation of two files (the view and the controller) and the modification of an existing file, the router.

The view

First off , we open a new blade view file named helloview.blade.php with the "Hello World" string. Create it in the directory app/resources/views

```
<h1>Hello, World</h1>
```

The controller

Now we create a controller that will manage the display of that view with the "Hello World" string. We'll use artisan in the command line.

```
$> cd your_laravel_project_root_directory
$> php artisan make:controller HelloController
```

That will just create a file (app/Http/Controllers/HelloController.php) containing the class that is

our new controller `HelloController`.

Edit that new file and write a new method `hello` that will display the view we created before.

```
public function hello()
{
        return view('helloview');
}
```

That 'helloview' argument in the view function is just the name of the view file without the trailing ".blade.php". Laravel will know how to find it.

Now when we call the method `hello`of the controller `HelloController`it will display the message. But how do we link that to a call to `http://mylaravel.com/helloworld` ? With the final step, the routing.

The router

Open the existing file `app/routes/web.php` (in older laravel versions `app/Http/routes.php`) and add this line:

```
Route::get('/helloworld', 'HelloController@hello');
```

which is a very self-explaining command saying to our laravel app: "When someone uses the `GET` verb to access '/helloworld' in this laravel app, return the results of calling the function `hello` in the `HelloController`controller.

Read Installation Guide online: https://riptutorial.com/laravel/topic/2187/installation-guide

Chapter 34: Introduction to laravel-5.2

Introduction

Laravel is a MVC framework with bundles, migrations, and Artisan CLI. Laravel offers a robust set of tools and an application architecture that incorporates many of the best features of frameworks like CodeIgniter, Yii, ASP.NET MVC, Ruby on Rails, Sinatra, and others. Laravel is an Open Source framework. It has a very rich set of features which will boost the speed of Web Development. If you familiar with Core PHP and Advanced PHP, Laravel will make your task easier. It will save a lot time.

Remarks

This section provides an overview of what laravel-5.1 is, and why a developer might want to use it.

It should also mention any large subjects within laravel-5.1, and link out to the related topics. Since the Documentation for laravel-5.1 is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Instructions on installing Laravel 5.1 on a Linux/Mac/Unix Machine.

Before initiating the installation, check if the following requirements are met:

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

Let's begin the installation:

1. Install composer. Composer Documentation
2. Run `composer create-project laravel/laravel <folder-name> "5.1.*"`
3. Ensure that the `storage` folder and the `bootstrap/cache` folder are writable.
4. Open the `.env` file and set the configuration information like database credentials, debug status, application environment, etc.
5. Run `php artisan serve` and point your browser to `http://localhost:8000`. If everything is fine then you should get the page

Install Laravel 5.1 Framework on Ubuntu 16.04, 14.04 & LinuxMint

Step 1 – Install LAMP

To start with Laravel, we first need to set up a running LAMP server. If you have already running LAMP stack skip this step else use followings commands to set up lamp on Ubuntu system.

Install PHP 5.6

```
$ sudo apt-get install python-software-properties
$ sudo add-apt-repository ppa:ondrej/php
$ sudo apt-get update
$ sudo apt-get install -y php5.6 php5.6-mcrypt php5.6-gd
```

Install Apache2

```
$ apt-get install apache2 libapache2-mod-php5
```

Install MySQL

```
$ apt-get install mysql-server php5.6-mysql
```

Step 2 – Install Composer

Composer is required for installing Laravel dependencies. So use below commands to download and use as a command in our system.

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
$ sudo chmod +x /usr/local/bin/composer
```

Step 3 – Install Laravel

To download latest version of Laravel, Use below command to clone master repo of laravel from github.

```
$ cd /var/www
$ git clone https://github.com/laravel/laravel.git
```

Navigate to Laravel code directory and use composer to install all dependencies required for Laravel framework.

```
$ cd /var/www/laravel
$ sudo composer install
```

Dependencies installation will take some time. After than set proper permissions on files.

```
$ chown -R www-data.www-data /var/www/laravel
$ chmod -R 755 /var/www/laravel
$ chmod -R 777 /var/www/laravel/app/storage
```

Step 4 – Set Encryption Key

Now set the 32 bit long random number encryption key, which used by the Illuminate encrypter service.

```
$ php artisan key:generate

Application key [uOHTNu3Au1Kt7Uloyr2Py9blU0J5XQ75] set successfully.
```

Now edit config/app.php configuration file and update above generated application key as followings. Also make sure cipher is set properly.

```
'key' => env('APP_KEY', 'uOHTNu3Au1Kt7Uloyr2Py9blU0J5XQ75'),

'cipher' => 'AES-256-CBC',
```

Step 5 – Create Apache VirtualHost

Now add a Virtual Host in your Apache configuration file to access Laravel framework from web browser. Create Apache configuration file under /etc/apache2/sites-available/ directory and add below content.

```
$ vim /etc/apache2/sites-available/laravel.example.com.conf
```

This is the Virtual Host file structure.

```
<VirtualHost *:80>

        ServerName laravel.example.com DocumentRoot
        /var/www/laravel/public

        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        <Directory /var/www/laravel>
                AllowOverride All
        </Directory>

        ErrorLog ${APACHE_LOG_DIR}/error.log LogLevel
        warn
```

Finally lets enable website and reload Apache service using below command.

```
$ a2ensite laravel.example.com
$ sudo service apache2 reload
```

Step 6 – Access Laravel

At this point you have successfully completed Laravel 5 PHP framework on your system. Now

make host file entry to access your Laravel application in web browser. Change `127.0.0.1` with your server ip and `laravel.example.com` with your domain name configured in Apache.

```
$ sudo echo "127.0.0.1               laravel.example.com" >> /etc/hosts
```

And access http://laravel.example.com in your favorite web browser as below.

Read Introduction to laravel-5.2 online: https://riptutorial.com/laravel/topic/1987/introduction-to-laravel-5-2

Chapter 35: Introduction to laravel-5.3

Introduction

New features, improvements and changes from Laravel 5.2 to 5.3

Examples

The $loop variable

It is known for a while that dealing with loops in Blade has been limited, as of 5.3 there is a variable called `$loop` available

```
@foreach($variables as $variable)

    // Within here the `$loop` variable becomes available

    // Current index, 0 based
    $loop->index;

    // Current iteration, 1 based
    $loop->iteration;

    // How many iterations are left for the loop to be complete
    $loop->remaining;

    // Get the amount of items in the loop
    $loop->count;

    // Check to see if it's the first iteration ...
    $loop->first;

    // ... Or last iteration
    $loop->last;

    //Depth of the loop, ie if a loop within a loop the depth would be 2, 1 based counting.
    $loop->depth;
```

Read Introduction to laravel-5.3 online: https://riptutorial.com/laravel/topic/9231/introduction-to-laravel-5-3

Chapter 36: Laravel Docker

Introduction

A challenge that every developer and development team faces is environment consistency. Laravel is one of the most popular PHP frameworks today. DDocker, on the other hand, is a virtualization method that eliminates *“works on my machine”* issues when cooperating on code with other developers. The two together create a fusion of **useful** and **powerful**. Although both of them do very different things, they can both be combined to create amazing products.

Examples

Using Laradock

Laradock is a project that provides a ready to go contains tailored for Laravel use.

Download or clone Laradock in your project's root folder:

```
git clone https://github.com/Laradock/laradock.git
```

Change directory into Laradock and generate the `.env` file needed to run your configurations:

```
cd laradock
cp .env-example .env
```

You are now ready to run docker. The first time you run the container it will download all the need packages from the internet.

```
docker-compose up -d nginx mysql redis beanstalkd
```

Now you can open your browser and view your project on `http://localhost`. For

the full Laradock documentation and configuration click here.

Read Laravel Docker online: https://riptutorial.com/laravel/topic/10034/laravel-docker

Chapter 37: Laravel Packages

Examples

laravel-ide-helper

This package generates a file that your IDE understands, so it can provide accurate autocompletion. Generation is done based on the files in your project.

Read more about this here

laravel-datatables

This package is created to handle server-side works of DataTables jQuery Plugin via AJAX option by using Eloquent ORM, Fluent Query Builder or Collection.

Read more about this here or here

Intervention Image

Intervention Image is an open source PHP image handling and manipulation library. It provides an easier and expressive way to create, edit, and compose images and supports currently the two most common image processing libraries GD Library and Imagick.

Read more about this here

Laravel generator

Get your APIs and Admin Panel ready in minutes.Laravel Generator to generate CRUD, APIs, Test Cases and Swagger Documentation

Read more about this here

Laravel Socialite

Laravel Socialite provides an expressive, fluent interface to OAuth authentication with Facebook, Twitter, Google, LinkedIn, GitHub and Bitbucket. It handles almost all of the boilerplate social authentication code you are dreading writing.

Read more about this here

Official Packages

Cashier

Laravel Cashier provides an expressive, fluent interface to Stripe's and Braintree's subscription

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

More about this package can be found here.

Envoy

Laravel Envoy provides a clean, minimal syntax for defining common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems.

This package can be found on Github.

Passport

Laravel already makes it easy to perform authentication via traditional login forms, but what about APIs? APIs typically use tokens to authenticate users and do not maintain session state between requests. Laravel makes API authentication a breeze using Laravel Passport, which provides a full OAuth2 server implementation for your Laravel application in a matter of minutes.

More about this package can be found here.

Scout

Laravel Scout provides a simple, driver-based solution for adding full-text search to your Eloquent models. Using model observers, Scout will automatically keep your search indexes in sync with your Eloquent records.

Currently, Scout ships with an Algolia driver; however, writing custom drivers is simple and you are free to extend Scout with your own search implementations.

More about this package can be found here.

Socialite

Laravel Socialite provides an expressive, fluent interface to OAuth authentication with Facebook, Twitter, Google, LinkedIn, GitHub and Bitbucket. It handles almost all of the boilerplate social authentication code you are dreading writing.

This package can be found on Github.

Read Laravel Packages online: https://riptutorial.com/laravel/topic/8001/laravel-packages

Chapter 38: lumen framework

Examples

Getting started with Lumen

The following example demonstrates using Lumen in WAMP / MAMP / LAMP environments.

To work with Lumen you need to setup couple of things first.

- Composer
- PHPUnit
- git (not required but strongly recommended)

Assuming you have all these three components installed (at least you need composer), first go to your web servers document root using terminal. MacOSX and Linux comes with a great terminal. You can use git bash (which is actually mingw32 or mingw64) in windows.

```
$ cd path/to/your/document/root
```

Then you need to use compose to install and create Lumen project. Run the following command.

```
$ composer create-project laravel/lumen=~5.2.0 --prefer-dist lumen-project
$ cd lumen-project
```

lumen-app in the code above is the folder name. You can change it as you like. Now you need to setup your virtual host to point to the path/to/document/root/lumen-project/public folder. Say you mapped http://lumen-project.local to this folder. Now if you go to this url you should see a message like following (depending on your installed Lumen version, in my case it was 5.4.4)-

```
Lumen (5.4.4) (Laravel Components 5.4.*)
```

If you open lumen-project/routers/web.php file there you should see the following-

```
$app->get('/', function () use($app) { return $app-
        >version();
});
```

Congratulations! Now you have a working Lumen installation. No you can extend this app to listen to your custom endpoints.

Read lumen framework online: https://riptutorial.com/laravel/topic/9221/lumen-framework

Chapter 39: Macros In Eloquent Relationship

Introduction

We have new features for Eloquent Relationship in Laravel version 5.4.8. We can fetch a single instance of a hasMany (it is just one example) relationship by define it at on place and it will works for all relationship

Examples

We can fetch one instance of hasMany relationship

In our AppServiceProvider.php

```
public function boot()
{
    HasMany::macro('toHasOne', function() { return new
            HasOne(
                    $this->query,
                    $this->parent,
                    $this->foreignKey,
                    $this->localKey
            );
    });
```

Suppose we have shop modal and we are getting the list of products which has purchased. Suppose we have allPurchased relationship for Shop modal

```
public function allPurchased()
{
        return $this->hasMany(Purchased::class);
}

public function lastPurchased()
{
        return $this->allPurchased()->latest()->toHasOne();
```

Read Macros In Eloquent Relationship online: https://riptutorial.com/laravel/topic/8998/macros-in-eloquent-relationship

Chapter 40: Mail

Examples

Basic example

You can configure Mail by just adding/changing these lines in the app's **.ENV** file with your email provider login details, for example for using it with gmail you can use:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=yourEmail@gmail.com
MAIL_PASSWORD=yourPassword
```

Then you can start sending emails using Mail, for example:

```
$variable = 'Hello world!'; // A variable which can be use inside email blade template. Mail::send('your.blade.file', ['variable' =>
$variable], function ($message) {
                $message->from('john@doe.com');
                $message->sender('john@doe.com');
                $message->to(foo@bar.com);
                $message->subject('Hello World');
```

Read Mail online: https://riptutorial.com/laravel/topic/8014/mail

Chapter 41: Middleware

Introduction

Middleware are classes, that can be assigned to one or more route, and are used to make actions in the early or final phases of the request cycle. We can think of them as a series of layers an HTTP request has to pass through while it's executed

Remarks

A "Before" middleware will executes before the controller action code; while a "After" middleware executes after the request is handled by the application

Examples

Defining a Middleware

To define a new middleware we have to create the middleware class:

```
class AuthenticationMiddleware
{
	//this method will execute when the middleware will be triggered public function handle (
	$request, Closure $next )
	{
		if ( ! Auth::user() )
		{
			return redirect('login');
		}

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

Then we have to register the middleware: if the middleware should be bind to all the routes of the application, we should add it to the middleware property of `app/Http/Kernel.php`:

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

while if we only want to associate the middleware to some of the routes, we can add it to `$routeMiddleware`

```
//register the middleware as a 'route middleware' giving it the name of 'custom_auth' protected $routeMiddleware = [
	'custom_auth' => \App\Http\Middleware\AuthenticationMiddleware::class
];
```

and then bind it to the single routes like this:

```
//bind the middleware to the admin_page route, so that it will be executed for that route Route::get('admin_page',
'AdminController@index')->middleware('custom_auth');
```

Before vs. After Middleware

An example of "before" middleware would be as follows:

```
<?php

namespace App\Http\Middleware; use

Closure;

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

                return $next($request);
```

while "after" middleware would look like this:

```
<?php

namespace App\Http\Middleware; use

Closure;

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

                // Perform action return

                $response;
        }
```

The key difference is in how the `$request` parameter is handled. If actions are performed before `$next($request)` that will happen before the controller code is executed while calling `$next($request)` first will lead to the actions being performed after the controller code is executed.

Route Middleware

Any middleware registered as `routeMiddleware` in `app/Http/Kernel.php`can be assigned to a route. There are a few different ways to assign middleware, but they all do the same.

```
Route::get('/admin', 'AdminController@index')->middleware('auth', 'admin'); Route::get('admin/profile', ['using' => 'AdminController@index', 'middleware' => 'auth']); Route::get('admin/profile', ['using' => 'AdminController@index', 'middleware' => ['auth', 'admin']);
```

In all the examples above, you can also pass fully qualified class names as middleware, regardless if it's been registered as a route middleware.

```
use App\Http\Middleware\CheckAdmin;
Route::get('/admin', 'AdminController@index')->middleware(CheckAdmin::class);
```

Read Middleware online: https://riptutorial.com/laravel/topic/3419/middleware

Chapter 42: Multiple DB Connections in Laravel

Examples

Initial Steps

Multiple database connections, of any type, can be defined inside the database configuration file (likely `app/config/database.php`). For instance, to pull data from 2 MySQL databases define them both separately:

```
<?php
return array(

    'default' => 'mysql', 'connections'

    => array(

        # Our primary database connection 'mysql' =>
        array(
            'driver'        => 'mysql', 'host'
                            => 'host1', 'database'
                            => 'database1',
            'username'      => 'user1', 'password'
                            => 'pass1' 'charset'
                            => 'utf8',
            'collation' => 'utf8_unicode_ci', 'prefix'   => '',
        ),

            'driver'        =>  'mysql',
            'host'          =>  'host2',
            'database'      =>  'database2',
            'username'      =>  'user2',
            'password'      =>  'pass2'
            'charset'       =>  'utf8',
            'collation'     =>  'utf8_unicode_ci',
            'prefix'        =>  '',
        ),
    ),
);
```

The default connection is still set to `mysql`. This means unless otherwise specified, the application uses the `mysql` connection.

Using Schema builder

Within the Schema Builder, use the Schema facade with any connection. Run the `connection()` method to specify which connection to use:

```
Schema::connection('mysql2')->create('some_table', function($table)
{
        $table->increments('id'):
});
```

Using DB query builder

Similar to Schema Builder, define a connection on the Query Builder:

```
$users = DB::connection('mysql2')->select(...);
```

Using Eloquent

There are multiple ways to define which connection to use in the Eloquent models. One way is to set the $connection variable in the model:

```
<?php

class SomeModel extends Eloquent { protected

        $connection = 'mysql2';
```

The connection can also be defined at runtime via the `setConnection` method.

```
<?php

class SomeController extends BaseController { public function

        someMethod()
        {
                $someModel = new SomeModel;

                $someModel->setConnection('mysql2');

                $something = $someModel->find(1); return

                $something;
        }
}
```

From Laravel Documentation

Each individual connection can be accessed via the connection method on the `DB` facade, even when there are multiple connections defined. The `name` passed to the `connection` method should correspond to one of the connections listed in the `config/database.php` configuration file:

```
$users = DB::connection('foo')->select(...);
```

The raw can also be accessed, underlying PDO instance using the getPdo method on a

connection instance:

```
$pdo = DB::connection()->getPdo();
```

https://laravel.com/docs/5.4/database#using-multiple-database-connections

Read Multiple DB Connections in Laravel online: https://riptutorial.com/laravel/topic/9605/multiple-db-connections-in-laravel

Chapter 43: Naming Files when uploading with Laravel on Windows

Parameters

Param/Function	Description
file-upload	name of the file <input> field
$sampleName	could also be dynamically generated string or the name of the file uploaded by the user
app_path()	is Laravel helper to provide the absolute path to the application
getCLientOriginalExtension()	Laravel wrapper to fetch the extension of the file uploaded by the user as it was on the user machine

Examples

Generating timestamped file names for files uploaded by users.

Below won't work on a Windows machine

```
$file = $request->file('file_upload');
$sampleName = 'UserUpload';
$destination = app_path() . '/myStorage/';
$fileName = $sampleName . '-' . date('Y-m-d-H:i:s') . '.' .
$file->getClientOriginalExtension();
$file->move($destination, $fileName);
```

It will throw an error like "Could no move file to /path..."

Why? - This works perfectly on a Ubuntu server
The reason is that on Windows colon ':' is not allowed in a filename where as it is allowed on linux. This is such a small thing that we may not notice it upfront and keep wondering that why a code which is running well on Ubuntu (Linux) is not working?
Our first hunch would be to check the file permissions and things like that but we may not notice that colon ':' is the culprit here.
So in order to upload files on Windows, **Do not use colon':'while generating timestamped filenames**, instead do something like below:

```
$filename = $sampleName . '-' . date('Y-m-d-H_i_s') . '.' . $file-
>getClientOriginalExtension();                    //ex output UserUpload-2016-02-18-11_25_43.xlsx
```

```
                                                    OR

$filename = $sampleName . '-' .date('Y-m-d H i s') . '.' . $file-
>getClientOriginalExtension();                      //ex output UserUpload-2016-02-18 11 25 43.xlsx

                                                    OR

$filename = $sampleName . '-'.date('Y-m-d_g-i-A').'.' . $file->getClientOriginalExtension();
```

Read Naming Files when uploading with Laravel on Windows online: https://riptutorial.com/laravel/topic/2629/naming-files-when-uploading-with-laravel-on-windows

Chapter 44: Observer

Examples

Creating an observer

Observers are used for listening to livecycle callbacks of a certain model in Laravel.
These listeners may listen to any of the following actions:

- creating
- created
- updating
- updated
- saving
- saved
- deleting
- deleted
- restoring
- restored

Here is an example of an observer.

UserObserver

```
<?php

namespace App\Observers;

/**
  * Observes the Users model
  */
class UserObserver
{
        /**
          *   Function will be triggerd when a user is updated
          *
          *   @param Users $model
          */
          public function updated($model)
          {
                  // execute your own code
```

As shown in the user observer, we listen to the updated action, however before this class actually listens to the user model we first need to register it inside the `EventServiceProvider`.

EventServiceProvider

```
<?php
```

```
namespace App\Providers;

use Illuminate\Contracts\Events\Dispatcher as DispatcherContract;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

use App\Models\Users;
use App\Observers\UserObserver;

/**
  * Event service provider class
  */
class EventServiceProvider extends ServiceProvider
{
        /**
          *   Boot function
          *
          *   @param DispatcherContract $events
          */
        public function boot(DispatcherContract $events)
        {
                parent::boot($events);

                // In this case we have a User model that we want to observe
                // We tell Laravel that the observer for the user model is the UserObserver Users::observe(new
```

Now that we have registered our observer, the updated function will be called every time after saving the user model.

Read Observer online: https://riptutorial.com/laravel/topic/7128/observer

Chapter 45: Pagination

Examples

Pagination in Laravel

In other frameworks pagination is headache. Laravel makes it breeze, it can generate pagination by adding few lines of code in Controller and View.

Basic Usage

There are many ways to paginate items, but the simplest one is using the paginate method on query builder or an Eloquent query. Laravel out of the box take care of setting limit and offset based on the current page being viewed by user. By default, the current page is detected by the value of ?page query string argument on the HTTP request. And for sure, this value is detected by Laravel automatically and insert into links generated by paginator.

Now let's say we want to call the `paginate`method on query. In our example the passed argument to paginate is the number of items you would like to display "per page". In our case, let say we want to display 10 items per page.

```
<?php

namespace App\Http\Controllers; use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
          *  Show all of the users for the application.
          *
          *  @return Response
          */
        public function index()
        {
                $users = DB::table('users')->paginate(10);

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

> **Note:** Currently, pagination operations that use a `groupBy` statement cannot be executed efficiently by Laravel. If you need to use a `groupBy` with a paginated result set, it is recommended that you query the database and create a paginator manually.

Simple Pagination

Let say you just want to display Next and Previous links on your pagination view. Laravel provides you this option by using `simplePaginate` method.

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

Displaying Results In A View

Now lets display the pagination in view. Actually when you call the `paginate` or `simplePaginate` methods on Eloquent query, you receive a paginator instance. When paginate method is called, you receive an instance of `Illuminate\Pagination\LengthAwarePaginator`, while when you call `simplePaginate` method, you receive an instance of `Illuminate\Pagination\Paginator`. These instances / objects comes with several methods that explaines the result set. Moreover, in addition to these helpers methods, the paginator instances are iterators and can be looped as an array.

Once you received the results, you can easily render the page links using blade

```
<div class="container"> @foreach ($users as
        $user)
                {{ $user->name }}
        @endforeach
</div>
```

The `links` method will automatically render the links to other pages in result set. Each of these links will contain the specific page number i.e `?page` query string variable. The HTML generated by the links method is perfectly compatible with the Bootstrap CSS framework.

Changing pagination views

While using laravel pagination you are free to use your own custom views.So,when calling the links method on a paginator instance, pass the view name as the first argument to the method like :

```
{{ $paginator->links('view.name') }}
```

or

You can customize the pagination views is by exporting them to your `resources/views/vendor` directory using the vendor:publish command:

```
php artisan vendor:publish --tag=laravel-pagination
```

This command will place the views in the `resources/views/vendor/pagination` directory. The `default.blade.php` file within this directory corresponds to the default pagination view. Edit this file to modify the HTML of pagination.

Read Pagination online: https://riptutorial.com/laravel/topic/2359/pagination

Chapter 46: Permissions for storage

Introduction

Laravel requires some folders to be writable for the web server user.

Examples

Example

We also need to set correct permissions for storage files in the server. So, we need to give a write permission in the storage directory as follows:

```
$ chmod -R 777 ./storage ./bootstrap
```

or you may use

```
$ sudo chmod -R 777 ./storage ./bootstrap
```

For windows

Make sure you are an admin user on that computer with writeable access

```
xampp\htdocs\laravel\app\storage needs to be writable
```

The NORMAL way to set permissions is to have your files owned by the webserver:

```
sudo chown -R www-data:www-data /path/to/your/root/directory
```

Read Permissions for storage online: https://riptutorial.com/laravel/topic/9797/permissions-for-storage

Chapter 47: Policies

Examples

Creating Policies

Since defining all of the authorization logic in the `AuthServiceProvider` could become cumbersome in large applications, Laravel allows you to split your authorization logic into "Policy" classes. Policies are plain PHP classes that group authorization logic based on the resource they authorize.

You may generate a policy using the `make:policy` artisan command. The generated policy will be placed in the `app/Policies` directory:

```
php artisan make:policy PostPolicy
```

Read Policies online: https://riptutorial.com/laravel/topic/7344/policies

Chapter 48: Queues

Introduction

Queues allow your application to reserve bits of work that are time consuming to be handled by a background process.

Examples

Use-cases

For example, if you are sending an email to a customer after starting a task, it's best to immediately redirect the user to the next page while queuing the email to be sent in the background. This will speed up the load time for the next page, since sending an email can sometimes take several seconds or longer.

Another example would be updating an inventory system after a customer checks out with their order. Rather than waiting for the API calls to complete, which may take several seconds, you can immediately redirect user to the checkout success page while queuing the API calls to happen in the background.

Queue Driver Configuration

Each of Laravel's queue drivers are configured from the `config/queue.php` file. A queue driver is the handler for managing how to run a queued job, identifying whether the jobs succeeded or failed, and trying the job again if configured to do so.

Out of the box, Laravel supports the following queue drivers:

sync

Sync, or synchronous, is the default queue driver which runs a queued job within your existing process. With this driver enabled, you effectively have no queue as the queued job runs immediately. This is useful for local or testing purposes, but clearly not recommended for production as it removes the performance benefit from setting up your queue.

database

This driver stores queued jobs in the database. Before enabling this driver, you will need to create database tables to store your queued and failed jobs:

```
php artisan queue:table php
artisan migrate
```

sqs

This queue driver uses Amazon's Simple Queue Service to manage queued jobs. Before enabling this job you must install the following composer package: `aws/aws-sdk-php ~3.0`

Also note that if you plan to use delays for queued jobs, Amazon SQS only supports a maximum delay of 15 minutes.

iron

This queue drivers uses Iron to manage queued jobs.

redis

This queue driver uses an instance of Redis to manage queued jobs. Before using this queue driver, you will need to configure a copy of Redis and install the following composer dependency: `predis/predis ~1.0`

beanstalkd

This queue driver uses an instance of Beanstalk to manage queued jobs. Before using this queue driver, you will need to configure a copy of Beanstalk and install the following composer dependency: `pda/pheanstalk ~3.0`

null

Specifying null as your queue driver will discard any queued jobs.

Read Queues online: https://riptutorial.com/laravel/topic/2651/queues

Chapter 49: Remove public from URL in laravel

Introduction

How to remove `public` from URL in Laravel, there are many answers on internet but the easiest way is described below

Examples

How to do that?

Follow these steps to remove `public` from the url

1. Copy `.htaccess`file from `/public`directory to `Laravel/project`root folder.
2. Rename the `server.php` in the `Laravel/project`root folder to `index.php`.

Cheers you will be good now.

Please Note: It is tested on *Laravel 4.2*, *Laravel 5.1*, *Laravel 5.2*, *Laravel 5.3*.

I think this is the easiest way to remove `public` from the url.

Remove the public from url

1. Renaming the `server.php` to `index.php`
2. Copy the `.htaccess`from `public`folder to `root`folder
3. Changing `.htaccess` a bit as follows for statics:

```
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)/$ /$1 [L,R=301]

RewriteCond %{REQUEST_URI} !(\.css|\.js|\.png|\.jpg|\.gif|robots\.txt)$ [NC] RewriteCond
%{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]

RewriteCond %{REQUEST_FILENAME} !-d RewriteCond
%{REQUEST_FILENAME} !-f RewriteCond
```

Sometimes I've use this method for removing `public` form url.

Read Remove public from URL in laravel online: https://riptutorial.com/laravel/topic/9786/remove-public-from-url-in-laravel

Chapter 50: Requests

Examples

Getting input

The primary way of getting input would be from injecting the `Illuminate\Http\Request` into your controller, after that there are numerous ways of accessing the data, 4 of which are in the example below.

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

Illuminate\Http\Request;

class UserController extends Controller
{
	public function store(Request $request)
	{
		// Returns the username value
		$name = $request->input('username');

		// Returns the username value
		$name = $request->username;

		// Returns the username value
		$name = request('username');

		// Returns the username value again
		$name = request()->username;
```

When using the `input` function it is also possible to add a default value for when the request input is not available

```
$name = $request->input('username', 'John Doe');
```

Read Requests online: https://riptutorial.com/laravel/topic/3076/requests

Chapter 51: Requests

Examples

Obtain an Instance of HTTP Request

To obtain an instance of an HTTP Request, class `Illuminate\Http\Request` need to be type hint either in the constructor or the method of the controller.

Example code:

```
 <?php

namespace App\Http\Controllers;

/* Here how we illuminate the request class in controller */ use
Illuminate\Http\Request;

use Illuminate\Routing\Controller; class PostController

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

        /*
        * so typecasting Request class in our method like above avails the
        * HTTP GET/POST/PUT etc method params in the controller to use and
        * manipulate
```

Request Instance with other Parameters from routes in controller method

Sometimes we need to accept route params as well as access the HTTP Request params. We can still type hint the Requests class in laravel controller and achieve that as explained below

E.g. We have a route that update a certain post like this (passing post id i route)

```
Route::put('post/{id}', 'PostController@update');
```

Also since user have edited other edit form fields, so that will be available in HTTP Request

Here is how to access both in our method

```
    public function update(Request $request,$id){
        //This way we have $id param from route and $request as an HTTP Request object

    }
```

Read Requests online: https://riptutorial.com/laravel/topic/4929/requests

Chapter 52: Route Model Binding

Examples

Implicit Binding

Laravel automatically resolves Eloquent models defined in routes or controller actions whose variable names match a route segment name. For example:

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

In this example, since the Eloquent $user variable defined on the route matches the {user} segment in the route's URI, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

If the model's table name is composed from multiple words, to make the implicit model binding working the input variable should be all lowercase;
For example, if the user can do some kind of *action*, and we want to access this action, the route will be:

```
Route::get('api/useractions/{useraction}', function (App\UserAction $useraction) { return $useraction->description;
});
```

Explicit Binding

To register an explicit binding, use the router's model method to specify the class for a given parameter. You should define your explicit model bindings in the boot method of the RouteServiceProvider class

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

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

Next, we can define a route that contains {user} parameter.

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

});
```

Since we have bound all `{user}` parameters to the `App\User`model, a User instance will be injected

into the route. So, for example, a request to `profile/1` will inject the User instance from the database which has an **ID** of **1**.

If a matching model instance is not found in the database, a **404 HTTP** response will be automatically generated.

Read Route Model Binding online: https://riptutorial.com/laravel/topic/7098/route-model-binding

Chapter 53: Routing

Examples

Basic Routing

Routing defines a map between HTTP methods and URIs on one side, and actions on the other. Routes are normally written in the `app/Http/routes.php` file.

In its simplest form, a route is defined by calling the corresponding HTTP method on the Route facade, passing as parameters a string that matches the URI (relative to the application root), and a callback.

For instance: a route to the root URI of the site that returns a view `home` looks like this:

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

A route for a post request which simply echoes the post variables:

```
Route::post('submit', function() { return
      Input::all();
});

//or

Route::post('submit', function(\Illuminate\Http\Request $request) { return $request->all();
});
```

Routes pointing to a Controller method

Instead of defining the callback inline, the route can refer to a controller method in [ControllerClassName@Method] syntax:

```
Route::get('login', 'LoginController@index');
```

A route for multiple verbs

The `match` method can be used to match an array of HTTP methods for a given route:

```
Route::match(['GET', 'POST'], '/', 'LoginController@index');
```

Also you can use `all` to match any HTTP method for a given route:

```
Route::all('login', 'LoginController@index');
```

Route Groups

Routes can be grouped to avoid code repetition.

Let's say all URIs with a prefix of `/admin` use a certain middleware called `admin` and they all live in the `App\Http\Controllers\Admin` namespace.

A clean way of representing this using Route Groups is as follows:

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

                // something.dev/admin
                // 'App\Http\Controllers\Admin\IndexController'
                // Uses admin middleware
                Route::get('/', ['uses' => 'IndexController@index']);

                // something.dev/admin/logs
                // 'App\Http\Controllers\Admin\LogsController'
                // Uses admin middleware
```

Named Route

Named routes are used to generate a URL or redirects to a specific route. The advantage of using a named route is, if we change the URI of a route in future, we wouldn't need to change the URL or redirects pointing to that route if we are using a named route. But if the links were generated using the url [eg. `url('/admin/login')`], then we would have to change everywhere where it is used.

Named routes are created using `as` array key

```
Route::get('login', ['as' => 'loginPage', 'uses' => 'LoginController@index']);
```

or using method `name`

```
Route::get('login', 'LoginController@index')->name('loginPage');
```

Generate URL using named route

To generate a url using the route's name

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

If you are using the route name for redirection

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

Route Parameters

You can use route parameters to get the part of the URI segment. You can define a optional or required route parameter/s while creating a route. Optional parameters have a `?` appended at the end of the parameter name. This name is enclosed in a curly braces `{}`

Optional Parameter

```
Route::get('profile/{id?}', ['as' => 'viewProfile', 'uses' => 'ProfileController@view']);
```

This route can be accessed by `domain.com/profile/23` where 23 is the id parameter. In this example the `id` is passed as a parameter in `view` method of `ProfileController`. Since this is a optional parameter accessing `domain.com/profile` works just fine.

Required Parameter

```
Route::get('profile/{id}', ['as' => 'viewProfile', 'uses' => 'ProfileController@view']);
```

Note that required parameter's name doesn't have a `?` at the end of the parameter name.

Accessing the parameter in controller

In your controller, your view method takes a parameter with the **same** name as the one in the `routes.php` and can be used like a normal variable. Laravel takes care of injecting the value:

```
public function view($id){ echo $id;
}
```

Catch all routes

If you want to catch all routes, then you could use a regular expression as shown:

```
Route::any('{catchall}', 'CatchAllController@handle')->where('catchall', '.*');
```

Important: If you have other routes and you don't want for the catch-all to interfere, you should put it in the end. For example:

Catching all routes except already defined

```
Route::get('login',                 'AuthController@login');
Route::get('logout', 'AuthController@logout'); Route::get('home',
                                    'HomeController@home');

// The catch-all will match anything except the previous defined routes. Route::any('{catchall}',
```

Routes are matched in the order they are declared

This is a common gotcha with Laravel routes. Routes are matched in the order that they are declared. The first matching route is the one that is used.

This example will work as expected:

```
Route::get('/posts/{postId}/comments/{commentId}', 'CommentController@show'); Route::get('/posts/{postId}',
'PostController@show');
```

A get request to `/posts/1/comments/1` will invoke `CommentController@show`. A get request to `/posts/1` will invoke `PostController@show`.

However, this example will not work in the same manner:

```
Route::get('/posts/{postId}', 'PostController@show'); Route::get('/posts/{postId}/comments/{commentId}',
'CommentController@show');
```

A get request to `/posts/1/comments/1` will invoke `PostController@show`. A get request to `/posts/1`will invoke `PostController@show`.

Because Laravel uses the first matched route, the request to `/posts/1/comments/1` matches `Route::get('/posts/{postId}', 'PostController@show');` and assigns the variable `$postId`to the value `1/comments/1`. This means that `CommentController@show` will never be invoked.

Case-insensitive routes

Routes in Laravel are case-sensitive. It means that a route like

```
Route::get('login', ...);
```

will match a GET request to `/login` but will not match a GET request to `/Login`.

In order to make your routes case-insensitive, you need to create a new validator class that will match requested URLs against defined routes. The only difference between the new validator and the existing one is that it will append the **i** modifier at the end of regular expression for the compiled route to switch enable case-insensitive matching.

```
<?php namespace Some\Namespace;

use Illuminate\Http\Request; use
Illuminate\Routing\Route;
use Illuminate\Routing\Matching\ValidatorInterface;

class CaseInsensitiveUriValidator implements ValidatorInterface
{
    public function matches(Route $route, Request $request)
    {
        $path = $request->path() == '/' ? '/' : '/'.$request->path();
        return preg_match(preg_replace('/$/','i', $route->getCompiled()->getRegex()), rawurldecode($path));
    }
```

In order for Laravel to use your new validator, you need to update the list of matchers that are used to match URL to a route and replace the original UriValidator with yours.

In order to do that, add the following at the top of your routes.php file:

```
<?php
use Illuminate\Routing\Route as IlluminateRoute; use
Your\Namespace\CaseInsensitiveUriValidator; use
Illuminate\Routing\Matching\UriValidator;

$validators = IlluminateRoute::getValidators();
$validators[] = new CaseInsensitiveUriValidator;
IlluminateRoute::$validators = array_filter($validators, function($validator) { return get_class($validator) !=
    UriValidator::class;
```

This will remove the original validator and add yours to the list of validators.

Read Routing online: https://riptutorial.com/laravel/topic/1284/routing

Chapter 54: Seeding

Remarks

Database seeding allows you to insert data, general test data into your database. By default there is a `DatabaseSeeder` class under `database/seeds`.

Running seeders can be done with

```
php artisan db:seed
```

Or if you only want to process a single class

```
php artisan db:seed --class=TestSeederClass
```

As with all artisan commands, you have access to a wide array of methods which can be found in the api documentation

Examples

Inserting data

There are several ways to insert data:

Using the DB Facade

```
public function run()
{
        DB::table('users')
            ->insert([
                'name' => 'Taylor', 'age'
                            => 21
         ]);
```

Via Instantiating a Model

```
public function run()
{
        $user = new User;
        $user->name = 'Taylor';
        $user->save();
}
```

Using the create method

```
public function run()
{
        User::create([
                'name' => 'Taylor', 'age'
                                => 21
        ]);
}
```

Using factory

```
public function run()
{
        factory(App\User::class, 10)->create();
}
```

Seeding && deleting old data and reseting auto-increment

```
public function run()
{
        DB::table('users')->delete();
        DB::unprepared('ALTER TABLE users AUTO_INCREMENT=1;');
        factory(App\User::class, 200)->create();
}
```

See the Persisting example for more information on inserting/updating data.

Calling other seeders

Within your `DatabaseSeeder` class you are able to call other seeders

```
$this->call(TestSeeder::class)
```

This allows you to keep one file where you can easily find your seeders. Keep in mind that you need to pay attention to the order of your calls regarding foreign key constraints. You can't reference a table that doesn't exist yet.

Creating a Seeder

To create seeders, you may use the `make:seeder` Artisan command. All seeders generated will be placed in the `database/seeds` directory.

```
$ php artisan make:seeder MoviesTableSeeder
```

Generated seeders will contain one method: `run`. You may insert data into your database in this method.

```
<?php

use Illuminate\Database\Seeder;
```

```
use Illuminate\Database\Eloquent\Model;

class MoviesTableSeeder extends Seeder
{
        /**
          *  Run the database seeds.
          *
          *  @return void
          */
        public function run()
        {
                App\Movie::create([
                        'name' => 'A New Hope', 'year' =>
                        '1977'
                ]);

                App\Movie::create([
                        'name' => 'The Empire Strikes Back', 'year' => '1980'
                ]);
```

You will generally want to call all your seeders inside the `DatabaseSeeder` class.

Once you're done writing the seeders, use the `db:seed` command. This will run `DatabaseSeeder`'s `run` function.

```
$ php artisan db:seed
```

You may also specify to run a specific seeder class to run individually using the `--class` option.

```
$ php artisan db:seed --class=UserSeeder
```

If you want to rollback and rerun all migrations, and then reseed:

```
$ php artisan migrate:refresh --seed
```

> The `migrate:refresh --seed` command is a shortcut to these 3 commands:
>
> ```
> $ php artisan migrate:reset      # rollback all migrations # run
> $ php artisan migrate            migrations
> $ php artisan db:seed
> ```

Safe reseeding

You may want to re-seed your database without affecting your previously created seeds. For this purpose, you can use `firstOrCreate` in your seeder:

```
EmployeeType::firstOrCreate([ 'type' =>
        'manager',
]);
```

Then you can seed the database:

```
php artisan db:seed
```

Later, if you want to add another type of employee, you can just add that new one in the same file:

```
EmployeeType::firstOrCreate([ 'type' =>
        'manager',
]);
EmployeeType::firstOrCreate([ 'type' =>
        'secretary',
```

And seed your database again with no problems:

```
php artisan db:seed
```

Notice in the first call you are retrieving the record but doing nothing with it.

Read Seeding online: https://riptutorial.com/laravel/topic/3272/seeding

Chapter 55: Services

Examples

Introduction

Laravel allows access to a variety of classes called Services. Some services are available out of the box, but you can create them by yourself.

A service can be used in multiple files of the application, like controllers. Let's imagine a Service `OurService` implementing a `getNumber()`method returning a random number:

```
class SomeController extends Controller {

	public function getRandomNumber()
	{
		return app(OurService::class)->getNumber();
	}
}
```

To create a Service, it is only needed to create a new Class:

```
# app/Services/OurService/OurService.php

<?php
namespace App\Services\OurService;

class OurService
{
	public function getNumber()
	{
		return rand();
	}
```

At this time, you could already use this service in a controller, but you would need to instantiate a new object each time you would need it:

```
class SomeController extends Controller {

	public function getRandomNumber()
	{
		$service = new OurService(); return $service-
		>getNumber();
	}

	public function getOtherRandomNumber()
	{
		$service = new OurService(); return
		$service->getNumber();
```

That is why the next step is to register your service into the **Service Container**. When you register you Service into the Service Container, you don't need to create a new object every time you need it.

To register a Service into the Service Container, you need to create a **Service Provider**. This Service Provider can:

1. Register your Service into the Service Container with *the register method*)
2. Injecting other Services into your Service (dependencies) with *the boot method*

A **Service Provider** is a class extending the abstract class `Illuminate\Support\ServiceProvider`. It needs to implement the `register()` method to **register** a Service into the **Service Container**:

```
# app/Services/OurService/OurServiceServiceProvider.php

<?php
namespace App\Services\OurService;

use Illuminate\Support\ServiceProvider;

class OurServiceServiceProvider extends ServiceProvider
{
	public function register()
	{
		$this->app->singleton('OurService', function($app) { return new
			OurService();
		});
```

All the **Service Providers** are saved in an array in `config/app.php`. So we need to register our Service Provider into this array:

```
return [

	...

	'providers' => [

		...

		App\Services\OurService\OurServiceServiceProvider::class,

		...

	],

	...
```

We can now access our Service in a controller. Three possibilities:

1. Dependency Injection:

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

use App\Services\OurService\OurService; class

SomeController extends Controller
{
        public function construct(OurService $our_service)
        {
                dd($our_service->getNumber());
        }
```

2. Access via the `app()` helper:

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

use App\Services\OurService\OurService; class

SomeController extends Controller
{
        public function getRandomNumber()
        {
                return app('OurService')->getNumber();
        }
```

Laravel provides Facades, imaginary classes that you can use in all of your projects and reflect a Service. To access your service more easily, you can create a Facade:

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


Randomisator;


class SomeController extends Controller
{
        public function getRandomNumber()
        {
                return Randomisator::getNumber();
        }
```

To create a new Facade, you need to create a new Class extending `Illuminate\Support\Facades\Facade`. This class needs to implement the `getFacadeAccessor()` method and return the name of a service registered by a **Service Provider**:

```
# app/Services/OurService/OurServiceFacade.php


<?php
namespace App\Services\OurService;

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

```
class OurServiceFacade extends Facade
{
	protected static function getFacadeAccessor()
	{
		return 'OurService';
	}
}
```

You also need to register your Facade into `config/app.php`:

```
return [

	...

	'aliases' => [

		....

		'Randomisator' => App\Services\OurService\OurServiceFacade::class,
	],
```

The Facade is now accessible anywhere in your project.

If you want to access your service from your views, you can create a helper function. Laravel ships with some helpers function out of the box, like the `auth()` function or the `view()` function. To create a helper function, create a new file:

```
# app/Services/OurService/helpers.php

if (! function_exists('randomisator')) {
	/**
	 *  Get the available OurService instance.
	 *
	 *  @return \App\ElMatella\FacebookLaravelSdk
	 */
	function randomisator()
	{
		return app('OurService');
	}
```

You also need to register this file, but in your `composer.json` file:

```
{

	...

	"autoload": {
		"files": [
			"app/Services/OurService/helpers.php"
		],

		...
```

```
}
```

You can now use this helper in a view:

```
<h1>Here is a random number: {{ randomisator()->getNumber() }}</h1>
```

Read Services online: https://riptutorial.com/laravel/topic/1907/services

Chapter 56: Services

Examples

Binding an Interface To Implementation

In a Service Provider `register` method we can bind an interface to an implementation:

```
public function register()
{
	App::bind( UserRepositoryInterface::class, EloquentUserRepository::class );
}
```

From now on, everytime the app will need an instance of `UserRepositoryInterface`, Laravel will auto inject a new instance of `EloquentUserRepository` :

```
//this will get back an instance of EloquentUserRepository
$repo = App::make( UserRepositoryInterface:class );
```

Binding an Instance

We can use the Service Container as a Registry by binding an instance of an object in it and get it back when we'll need it:

```
// Create an instance.
$john = new User('John');

// Bind it to the service container.
App::instance('the-user', $john);

// ...somewhere and/or in another class...

// Get back the instance
```

Binding a Singleton to the Service Container

We can bind a class as a Singleton:

```
public function register()
{
	App::singleton('my-database', function()
	{
		return new Database();
	});
}
```

This way, the first time an instance of `'my-database'` will be requested to the service container, a new instance will be created. All the successive requests of this class will get back the first created

instance:

```
//a new instance of Database is created
$db = App::make('my-database');

//the same instance created before is returned
$anotherDb = App::make('my-database');
```

Introduction

The **Service Container** is the main Application object. It can be used as a Dependency Injection Container, and a Registry for the application by defining bindings in the Service Providers

Service Providers are classes where we define the way our service classes will be created through the application, bootstrap their configuration, and bind interfaces to implementations

Services are classes that wrap one or more logic correlated tasks together

Using the Service Container as a Dependency Injection Container

We can use the Service Container as a Dependency Injection Container by binding the creation process of objects with their dependencies in one point of the application

Let's suppose that the creation of a `PdfCreator` needs two objects as dependencies; every time we need to build an instance of `PdfCreator`, we should pass these dependencies to che constructor.
By using the Service Container as DIC, we define the creation of `PdfCreator` in the binding definition, taking the required dependency directly from the Service Container:

```
App:bind('pdf-creator', function($app) {

    // Get the needed dependencies from the service container.
    $pdfRender = $app->make('pdf-render');
    $templateManager = $app->make('template-manager');

    // Create the instance passing the needed dependencies. return new PdfCreator(
    $pdfRender, $templateManager );
```

Then, in every point of our app, to get a new `PdfCreator`, we can simply do:

```
$pdfCreator = App::make('pdf-creator');
```

And the Service container will create a new instance, along with the needed dependencies for us.

Read Services online: https://riptutorial.com/laravel/topic/1908/services

Chapter 57: Socialite

Examples

Installation

```
composer require laravel/socialite
```

This installation assumes you're using Composer for managing your dependencies with Laravel, which is a great way to deal with it.

Configuration

In your `config\services.php` you can add the following code

```
'facebook' => [
        'client_id' => 'your-facebook-app-id', 'client_secret' => 'your-
        facebook-app-secret', 'redirect' => 'http://your-callback-url',
],
```

You'll also need to add the Provider to your `config\app.php`

Look for `'providers' => []` array and, at the end of it, add the following

```
'providers' => [
            ...

            Laravel\Socialite\SocialiteServiceProvider::class,
]
```

A Facade is also provided with the package. If you would like to make usage of it make sure that the `aliases` array (also in your `config\app.php`) has the following code

```
'aliases' => [
        ....
        'Socialite' => Laravel\Socialite\Facades\Socialite::class,
]
```

Basic Usage - Facade

```
return Socialite::driver('facebook')->redirect();
```

This will redirect an incoming request to the appropriate URL to be authenticated. A basic example would be in a controller

```
<?php
```

```
namespace App\Http\Controllers\Auth; use

Socialite;

class AuthenticationController extends Controller {

        /**
          *   Redirects the User to the Facebook page to get authorization.
          *
          *   @return Response
          */
        public function facebook() {
                return Socialite::driver('facebook')->redirect();
        }
```

make sure your `app\Http\routes.php` file has a route to allow an incoming request here.

```
Route::get('facebook', 'App\Http\Controllers\Auth\AuthenticationController@facebook');
```

Basic Usage - Dependency Injection

```
/**
  *   LoginController constructor.
  *   @param Socialite $socialite
  */
public function construct(Socialite $socialite) {
        $this->socialite = $socialite;
}
```

Within the constructor of your Controller, you're now able to inject the `Socialite` class that will help you handle login with social networks. This will replace the usage of the Facade.

```
/**
  *   Redirects the User to the Facebook page to get authorization.
  *
  *   @return Response
  */
public function facebook() {
        return $this->socialite->driver('facebook')->redirect();
}
```

Socialite for API - Stateless

```
public function facebook() {
        return $this->socialite->driver('facebook')->stateless()->redirect()->getTargetUrl();
}
```

This will return the URL that the consumer of the API must provide to the end user to get authorization from Facebook.

Read Socialite online: https://riptutorial.com/laravel/topic/1312/socialite

Chapter 58: Sparkpost integration with Laravel 5.4

Introduction

Laravel 5.4 comes preinstalled with sparkpost api lib. Sparkpost lib requires secret key which one can find from their sparkpost account.

Examples

SAMPLE .env file data

To successfully create a sparkpost email api setup, add the below details to env file and your application will be good to start sending emails.

MAIL_DRIVER=sparkpost
SPARKPOST_SECRET=

NOTE: The above details does not give you the code written in controller which has the business logic to send emails using laravels Mail::send function.

Read Sparkpost integration with Laravel 5.4 online:
https://riptutorial.com/laravel/topic/10136/sparkpost-integration-with-laravel-5-4

Chapter 59: Task Scheduling

Examples

Creating a task

You can create a task (Console Command) in Laravel using Artisan. From your command line:

```
php artisan make:console MyTaskName
```

This creates the file in **app/Console/Commands/MyTaskName.php**. It will look like this:

```
<?php

namespace App\Console\Commands; use

Illuminate\Console\Command; class

MyTaskName extends Command
{
        /**
          *  The name and signature of the console command.
          *
          *  @var string
          */
        protected $signature = 'command:name';

        /**
          *  The console command description.
          *
          *  @var string
          */
        protected $description = 'Command description';

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

        /**
          *  Execute the console command.
          *
          *  @return mixed
          */
```

Some important parts of this definition are:

- The `$signature` property is what identifies your command. You will be able to execute this command later through the command line using Artisan by running `php artisan command:name` (Where `command:name` matches your command's `$signature`)
- The `$description` property is Artisan's help/usage displays next to your command when it is made available.
- The `handle()` method is where you write the code for your command.

Eventually, your task will be made available to the command line through Artisan. The `protected $signature = 'command:name';` property on this class is what you would use to run it.

Making a task available

You can make a task available to Artisan and to your application in the **app/Console/Kernel.php** file.

The `Kernel` class contains an array named `$commands` which make your commands available to your application.

Add your command to this array, in order to make it available to Artisan and your application.

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        Commands\Inspire::class,
        Commands\MyTaskName::class // This line makes MyTaskName available
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
```

Once this is done, you can now access your command via the command line, using Artisan. Assuming that your command has the `$signature` property set to `my:task`, you can run the following

command to execute your task:

```
php artisan my:task
```

Scheduling your task

When your command is made available to your application, you can use Laravel to schedule it to run at pre-defined intervals, just like you would a CRON.

In The **app/Console/Kernel.php** file you will find a `schedule`method that you can use to schedule your task.

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
        /**
          *  The Artisan commands provided by your application.
          *
          *  @var array
          */
        protected $commands = [
                Commands\Inspire::class,
                Commands\MyTaskName::class
        ];

        /**
          *  Define the application's command schedule.
          *
          *  @param     \Illuminate\Console\Scheduling\Schedule                    $schedule
          *  @return void
          */
        protected function schedule(Schedule $schedule)
        {
                $schedule->command('my:task')->everyMinute();
                // $schedule->command('my:task')->everyFiveMinutes();
                // $schedule->command('my:task')->daily();
```

Assuming your task's `$signature` is `my:task`you can schedule it as shown above, using the `Schedule $schedule` object. Laravel provides loads of different ways to schedule your command, as shown in the commented out lines above.

Setting the scheduler to run

The scheduler can be run using the command:

```
php artisan schedule:run
```

The scheduler needs to be run every minute in order to work correctly. You can set this up by creating a cron job with the following line, which runs the scheduler every minute in the background.

```
* * * * * php /path/to/artisan schedule:run >> /dev/null 2>&1
```

Read Task Scheduling online: https://riptutorial.com/laravel/topic/4026/task-scheduling

Chapter 60: Testing

Examples

Introduction

Writing testable code is an important part of building a robust, maintainable, and agile project. Support for PHP's most widely used testing framework, PHPUnit, is built right into Laravel. PHPUnit is configured using the `phpunit.xml` file, which resides in the root directory of every new Laravel application.

The `tests` directory, also in the root directory, contains the individual testing files which hold the logic for testing each portion of your application. Of course, it is your responsibility as a developer to write these tests as you build your application, but Laravel includes an example file, `ExampleTest.php`, to get you going.

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware; use
Illuminate\Foundation\Testing\DatabaseMigrations; use
Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
        /**
          *  A basic functional test example.
          *
          *  @return void
          */
        public function testBasicExample()
        {
                $this->visit('/')
                        ->see('Laravel 5');
```

In the `testBasicExample()` method, we visit the site's index page and make sure we see the text `Laravel 5` somewhere on that page. If the text is not present, the test will fail and generate an error.

Test without middleware and with a fresh database

To make artisan migrate a fresh database before running tests, `use DatabaseMigrations`. Also if you want to avoid middleware like Auth, `use WithoutMiddleware`.

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware; use
Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends TestCase
```

```
    use DatabaseMigrations, WithoutMiddleware;

    /**
     *  A basic functional test example.
     *
     *  @return void
     */
    public function testExampleIndex()
    {
        $this->visit('/protected-page')
          ->see('All good');
    }
```

Database transactions for mutliple database connection

DatabaseTransactions trait allows databases to rollback all the change during the tests. If you want to rollback multiple databases , you need to set $connectionsToTransact properties

```
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends TestCase
{
      use DatabaseTransactions;

      $connectionsToTransact =["mysql","sqlite"] //tell Laravel which database need to rollBack public function testExampleIndex()
    {
        $this->visit('/action/parameter')
          ->see('items');
    }
}
```

Testing setup, using in memory database

Following setup ensures that testing framework (PHPUnit) uses :memory: database.

config/database.php

```
'connections' => [

    'sqlite_testing' => [ 'driver'        => 'sqlite',
        'database' => ':memory:',
        'prefix'               => '',
    ],
    .
    .
    .
```

./phpunit.xml

```
    .
    .
```

```
        .
    </filter>
    <php>
        <env name="APP_ENV" value="testing"/>
        <env name="APP_URL" value="http://example.dev"/>
        <env name="CACHE_DRIVER" value="array"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="QUEUE_DRIVER" value="sync"/>
        <env name="DB_CONNECTION" value="sqlite_testing"/>
      </php>
```

Configuration

The phpunit.xml file is the default configuration file for tests and is already setup for testing with PHPUnit.

The default testing environment `APP_ENV` is defined as `testing` with `array` being the cache driver `CACHE_DRIVER`. With this setup, no data (session/cache) will be retained while testing.

To run tests against a specific environment like homestead the defaults can be changed to:

```
<env name="DB_HOST" value="192.168.10.10"/>
<env name="DB_DATABASE" value="homestead"/>
<env name="DB_USERNAME" value="homestead"/>
<env name="DB_PASSWORD" value="secret"/>
```

Or to use a temporary *in memory* database:

```
<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>
```

One last note to keep in mind from the Laravel documentation:

> Make sure to clear your configuration cache using the `config:clear` Artisan command before running your tests!

Read Testing online: https://riptutorial.com/laravel/topic/1249/testing

Chapter 61: Token Mismatch Error in AJAX

Introduction

I have analyzed that ratio of getting TokenMismatch Error is very high. And this error occurs because of some silly mistakes. There are many reasons where developers are making mistakes. Here are some of the examples i.e No _token on headers, No _token passed data when using Ajax, permission issue on storage path, an invalid session storage path.

Examples

Setup Token on Header

Set the token on `<head>` of your `default.blade.php`.

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

Add `ajaxSetup` on the top of your script, that will be accessible to everywhere. This will set headers on each `ajax` call

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

Set token on tag

Add below function to your `<form>` tag. This function will generate a hidden field named `_token`and filled value with the token.

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

Add `csrf_token ()` function to your hidden `_token`in the value attribute. This will generate only encrypted string.

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

Check session storage path & permission

Here I assume that project app url is `APP_URL=http://project.dev/ts/toys-store`

1. Set the writable permission to `storage_path('framework/sessions')` the folder.
2. Check the path of your laravel project `'path' => '/ts/toys-store',` the root of your laravel

project.

3. Change the name of your cookie `'cookie' => 'toys-store',`

```
return [
        'driver' => env('SESSION_DRIVER', 'file'), 'lifetime' => 120,
        'expire_on_close' => false, 'encrypt' =>
        false,
        'files' => storage_path('framework/sessions'), 'connection' => null,
        'table' => 'sessions', 'lottery' => [2, 100],
        'cookie' => 'toys-store', 'path' =>
        '/ts/toys-store', 'domain' => null,
        'secure' => false, 'http_only'
        => true,
];
```

Use _token field on Ajax

There are many ways to send `_token` on AJAX call

1. Get all input field's value within `<form>` tag using `var formData = newFormData($("#cart- add")[0]);`
2. Use `$("form").serialize();` or `$("form").serializeArray();`
3. Add `_token` manually on `data` of Ajax. using `$('meta[name="csrf-token"]').attr('content')` or `$('input[name="_token"]').val()`.
4. We can set as header on a particular Ajax call like below code.

```
$.ajax({
        url: $("#category-add").attr("action"), type: "POST",
        data: formData,
        processData: false,
        contentType: false,
        dataType: "json", headers: {
                'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
        }
});
```

Read Token Mismatch Error in AJAX online: https://riptutorial.com/laravel/topic/10656/token-mismatch-error-in-ajax

Chapter 62: use fields aliases in Eloquent

Read use fields aliases in Eloquent online: https://riptutorial.com/laravel/topic/7927/use-fields-aliases-in-eloquent

Chapter 63: Useful links

Introduction

In this topic, you can find useful links to improve your Laravel skills or extend your knowledge.

Examples

Laravel Ecosystem

- Laravel Scout - Laravel Scout provides a simple, driver-based solution for adding full-text search to your Eloquent models.
- Laravel Passport - API authentication without a headache. Passport is an OAuth2 server that's ready in minutes.
- Homestead - The official Laravel development environment. Powered by Vagrant, Homestead gets your entire team on the same page with the latest PHP, MySQL, Postgres, Redis, and more.
- Laravel Cashier - Make subscription billing painless with built-in Stripe and Braintree integrations. Coupons, swapping subscriptions, cancellations, and even PDF invoices are ready out of the box.
- Forge - Provision and deploy unlimited PHP applications on DigitalOcean, Linode, & AWS.
- Envoyer - Zero Downtime PHP Deployment.
- Valet - A Laravel development environment for Mac minimalists. No Vagrant, no Apache, no fuss.
- Spark - Powerful SaaS application scaffolding. Stop writing boilerplate & focus on your application.
- Lumen - If all you need is an API and lightning fast speed, try Lumen. It’s Laravel super-light.
- Statamic - A true CMS designed to make agencies profitable, developers happy, and clients hug you.

Education

- Laracasts - Learn practical, modern web development, through expert screencasts.
- Laravel News - Stay up to date with Laravel with Laravel News.
- Laravel.io - Forum with open-source code.

Podcasts

- Laravel News Podcasts
- The Laravel Podcasts

Read Useful links online: https://riptutorial.com/laravel/topic/9957/useful-links

Chapter 64: Valet

Introduction

Valet is a development environment tailor made for macOS. It abstracts away the need for virtual machines, Homestead, or Vagrant. No need to constantly update your `/etc/hosts` file anymore. You can even share your sites publicly using local tunnels.

Laravel Valet makes all sites available on a `*.dev` domain by binding folder names to domain names.

Syntax

- valet command [options] [arguments]

Parameters

Parameter	Values Set
command	domain, fetch-share-url, forget, help, install, link, links, list, logs, on-latest-version, open, park, paths, restart, secure, start, stop, uninstall, unlink, unsecure, which
options	-h, --help, -q, --quiet, -V, --version, --ansi, --no-ansi, -n, --no-interaction, -v, -vv, -vvv,--verbose
arguments	*(optional)*

Remarks

Because Valet for Linux and Windows are unofficial, there will not be support outside of their respective Github repositories.

Examples

Valet link

This command is useful if you want to serve a single site in a directory and not the entire directory.

```
cd ~/Projects/my-blog/ valet link
awesome-blog
```

Valet will create a symbolic link in `~/.valet/Sites` which points to your current working directory.

After running the link command, you can access the site in your browser at `http://awesome-blog.dev`.

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

Valet park

```
cd ~/Projects valet
park
```

This command will register your current working directory as a path that Valet should search for sites. Now, any Laravel project you create within your "parked" directory will automatically be served using the `http://folder-name.dev` convention.

Valet links

This command will display all the registered Valet links you have created and their corresponding file paths on your computer.

Command:

```
valet links
```

Sample Output:

```
...
site1 -> /path/to/site/one site2 ->
/path/to/site/two
...
```

Note 1: You can run this command from anywhere not just from within a linked folder.

Note 2: Sites will be listed without the ending **.dev** but you'll still use **site1.dev** to access your application from the browser.

Installation

***IMPORTANT!!* Valet is a tool designed for macOS only.**

Prerequisites

- Valet utilizes your local machine's HTTP port (port 80), therefore, you will not be able to use if *Apache* or *Nginx* are installed and running on the same machine.
- macOS' unofficial package manager Homebrew is required to properly use Valet.
- Make sure Homebrew is updated to the latest version by running `brew update` in the terminal.

Installation

- Install PHP 7.1 using Homebrew via `brew install homebrew/php/php71`.
- Install Valet with Composer via `composer global require laravel/valet`.
- Append `~/.composer/vendor/bin` directory to your system's "PATH" if it is not already there.
- Run the `valet install` command.

Post Install During the installation process, Valet installed *DnsMasq*. It also registered Valet's daemon to automatically launch when your system starts, so you don't need to run `valet start` or `valet install` every time you reboot your machine.

Valet domain

This command allows you to change or view the TLD *(top-level domain)* used to bind domains to your local machine.

Get The Current TLD

```
$ valet domain
> dev
```

Set the TLD

```
$ valet domain local
> Your Valet domain has been updated to [local].
```

Installation (Linux)

IMPORTANT!! Valet is a tool designed for macOS, the version below is ported for Linux OS.

Prerequisites

- **Do not** install valet as `root` or by using the `sudo` command.
- Valet utilizes your local machine's HTTP port (port 80), therefore, you will not be able to use if Apache or Nginx are installed and running on the same machine.
- An up to date version of `composer` is required to install and run Valet.

Installation

- Run `composer global require cpriego/valet-linux` to install Valet globally.
- Run the `valet install` command to finish the installation.

Post Install

During the installation process, Valet installed DnsMasq. It also registered Valet's daemon to automatically launch when your system starts, so you don't need to run `valet start` or `valet install` every time you reboot your machine.

The Official Documentation can be found here.

Read Valet online: https://riptutorial.com/laravel/topic/1906/valet

Chapter 65: Validation

Parameters

Parameter	Details
required	The field is required
sometimes	Run validation checks against a field only if that field is present in the input array
email	The input is a valid email
max:value	The input value should be below the maximum value
unique:db_table_name	The input value should be unique in the provided database table name
accepted	Yes / On / 1 true, useful for checking TOS
active_url	Must be a valid URL according to checkdnsrr
after *:date*	Field under validation must provide a value after the given date
alpha	The field under validation must be entirely alphabetic characters.
alpha_dash	The field under validation may have alpha-numeric characters, as well as dashes and underscores.
alpha_num	The field under validation must be entirely alpha-numeric characters.
array	Must be a PHP array
before *:date*	The field must be a value under the given date
between:min,max	The input value should be in between minimum (min) and maximum (max) value
boolean	The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.
confirmed	The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.
date	The field under validation must be a valid date according to the

Parameter	Details
	strtotime PHP function.
integer	The field under validation must be an integer
string	The field under validation must be a string type.

Examples

Basic Example

You can validate request data using the `validate` method (available in the base Controller, provided by the `ValidatesRequests` trait).

If the rules pass, your code will keep executing normally; however, if validation fails, an error response containing the validation errors will automatically be sent back:

- for typical HTML form requests, the user will be redirected to the previous page, with the form keeping the submitted values
- for requests that expect a JSON response, a HTTP response with code 422 will be generated

For example, in your `UserController`, you might be saving a new user in the `store` method, which would need validation before saving.

```
/**
  *  @param    Request      $request
  *  @return Response
  */
public function store(Request $request) {
        $this->validate($request, [ 'name' =>
                'required',
                'email' => 'email|unique:users|max:255'
        ],
        // second array of validation messages can be passed here [
                'name.required' => 'Please provide a valid name!', 'email.required' => 'Please
                provide a valid email!',
        ]);

        // The validation passed
```

In the example above, we validate that the `name` field exists with non-empty value. Secondly, we check that the `email` field has a valid e-mail format, is unique in the database table "users", and has maximum length of 255 characters.

> The `|` (pipe) character combines different validation rules for one field.

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
$this->validate($request, [ 'name' =>
        'bail|required',
        'email' => 'email|unique:users|max:255'
]);
```

The complete list of available validation rules can be found in the parameters section below.

Array Validation

Validating array form input fields is very simple.

Suppose you have to validate each name, email and father name in a given array. You could do the following:

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

if ($validator->fails()) {
        return back()->withInput()->withErrors($validator->errors());
}
```

Laravel displays default messages for validation. However, if you want custom messages for array based fields, you can add the following code:

```
[
        'name.*' => [
                'required' => 'Name field is required',
        ],
        'email.*' => [
                'unique'          => 'Unique Email is required',
        ],
        'fatherName.*' => [
                'required' => 'Father Name required',
        ]
```

Your final code will look like this:

```
$validator = \Validator::make($request->all(), ['name.*'    => 'required',
        'email.*'                => 'email|unique:users',
        'fatherName.*' => 'required',
], [
        'name.*'                 => 'Name Required',
        'email.*'                => 'Unique Email is required',
        'fatherName.*' => 'Father Name required',
]);

if ($validator->fails()) {
        return back()->withInput()->withErrors($validator->errors());
```

Other Validation Approaches

1) Form Request Validation

You may create a "form request" which can hold the authorization logic, validation rules, and error messages for a particular request in your application.

The `make:request` Artisan CLI command generates the class and places it in the `app/Http/Requests` directory:

```
php artisan make:request StoreBlogPostRequest
```

The `authorize` method can be overridden with the authorization logic for this request:

```
public function authorize()
{
	return $this->user()->can('post');
}
```

The `rules` method can be overridden with the specific rules for this request:

```
public function rules()
{
	return [
		'title' => 'required|unique:posts|max:255', 'body' => 'required',
	];
}
```

The `messages` method can be overridden with the specific messages for this request:

```
public function messages()
{
	return [
		'title.required' => 'A title is required',
		'title.unique' => 'There is another post with the same title', 'title.max' => 'The title may not
		exceed :max characters', 'body.required' => 'A message is required',
	];
}
```

In order to validate the request, just type-hint the specific request class on the corresponding controller method. If validation fails, an error response will be sent back.

```
public function store(StoreBlogPostRequest $request)
{
	// validation passed
}
```

---

2) Manually Creating Validators

For more flexibility, you may want to create a Validator manually, and handle the failed validation directly:

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

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

class PostController extends Controller
{
        public function store(Request $request)
        {
                $validator = Validator::make($request->all(), [ 'title' =>
                        'required|unique:posts|max:255', 'body' => 'required',
                ]);

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

                // Store the blog post...
```

2) Fluently creating rules

Occasionally you might need to create unique rules on the fly, working with the `boot()` method within a Service Provider might be over the top, as of Laravel 5.4 you can create new rules fluently by using the `Rule` class.

As an example we are going to work with the `UserRequest` for when you want to insert or update a user. For now we want a name to be required and the email address must be unique. The problem with using the `unique` rule is that if you are editing a user, they might keep the same email, so you need to exclude the current user from the rule. The following example shows how you can easily do this by utilising the new `Rule` class.

```
<?php
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest; use
Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class UserRequest extends FormRequest
{
        /**
          *  Determine if the user is authorized to make this request.
          *
          *  @return bool
          */
```

```
            return true;
        }

        /**
         * Get the validation rules that apply to the request.
         *
         * @return array
         */
        public function rules(Request $request)
        {

            return [
                'name'                  =>  'required',

                // Notice the value is an array and not a string like usual 'email'     =>      [
                    'required',
                    Rule::unique('users')->ignore($id)
                ]
            ];
        }
}
```

Single Form Request Class for POST, PUT, PATCH

Following the 'Form Request Validation' example, the same Request Class can be used for POST, PUT, PATCH so you do not have to create another class using the same/similar validations. This comes in handy if you have attributes in your table that are unique.

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules() { switch($this-
    >method()) {
        case 'GET':
        case 'DELETE':
            return [];
        case 'POST':
            return [
                'name'          => 'required|max:75|unique', 'category'
                => 'required',
                'price'         => 'required|between:0,1000',
            ];
        case 'PUT':
        case 'PATCH':
            return [
                'name'          => 'required|max:75|unique:product,name,' . $this->product, 'category' => 'required',
                'price'         => 'required|between:0,1000',
            ];
        default:break;
```

Starting from the top, our switch statement is going to look at the method type of the request (GET,

DELETE, POST, PUT, PATCH).

Depending on the method will return the array of rules defined. If you have a field that is unique, such as the name field in the example, you need to specify a particular id for the validation to ignore.

```
'field_name' => 'unique:table_name,column_name,' . $idToIgnore`
```

If you have a primary key labeled something other than id, you will specify the primary key column as the fourth parameter.

```
'field_name' => 'unique:table_name,column_name,' . $idToIgnore . ',primary_key_column'
```

In this example, we are using PUT and passing to the route (admin/products/{product}) the value of the product id. So $this->product will be equal to the idto ignore.

Now your PUT|PATCH and POSTvalidation rules do not need to be the same. Define your logic that fits your requirements. This technique allows you to reuse the custom messages you may have defined within the custom Form Request Class.

Error messages

Customizing error messages

The /resources/lang/[lang]/validation.php files contain the error messages to be used by the validator. You can edit them as needed.

Most of them have placeholders which will be automatically replaced when generating the error message.

For example, in 'required' => 'The :attribute field is required.', the :attribute placeholder will be replaced by the field name (alternatively, you can also customize the display value of each field in the attributes array in the same file).

Example

message configuration:

```
'required' => 'Please inform your :attribute.',
//... 'attributes => [
        'email' => 'E-Mail address'
]
```

rules:

```
`email' => `required`
```

resulting error message:

"Please inform your E-Mail address."

Customising error messages within a Request class

The Request class has access to a `messages()` method which should return an array, this can be used to override messages without having to go into the lang files. For example if we have a custom `file_exists` validation you can messages like below.

```
class SampleRequest extends Request {

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'image' =>          'required|file_exists'
        ];
    }

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    public function messages()
    {
        return [
            'image.file_exists' =>              'That file no longer exists or is invalid'
```

Displaying error messages

The validation errors are flashed to the session, and are also available in the `$errors`variable, which is automatically shared to all views.

Example of displaying the errors in a Blade view:

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
```

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

Custom Validation Rules

If you want to create a custom validation rule, you can do so for instance in the `boot` method of a service provider, via the Validator facade.

```
<?php
namespace App\Providers;

use Illuminate\Support\ServiceProvider; use Validator;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        Validator::extend('starts_with', function($attribute, $value, $parameters, $validator)
{
            return \Illuminate\Support\Str::startsWith($value, $parameters[0]);
        });

        Validator::replacer('starts_with', function($message, $attribute, $rule, $parameters)
{
            return str_replace(':needle', $parameters[0], $message);
        });
    }
```

The `extend` method takes a string which will be the name of the rule and a function which in turn will be passed the name of the attribute, the value being validated, an array of the rule parameters, and the validator instance, and should return whether the validation passes. In this example, we are checking if the value string starts with a given substring.

The error message for this custom rule can be set as usual in the `/resources/lang/[lang]/validation.php` file, and can contain placeholders, for instance, for parameters values:

```
'starts_with' => 'The :attribute must start with :needle.'
```

The `replacer` method takes a string which is the name of the rule and a function which in turn will be passed the original message (before replacing), the name of the attribute, the name of the rule, and an array of the rule parameters, and should return the message after replacing the placeholders as needed.

Use this rule as any other:

```
$this->validate($request, [
        'phone_number' => 'required|starts_with:+'
]);
```

Read Validation online: https://riptutorial.com/laravel/topic/1310/validation

Credits

S. No	Chapters	Contributors
1	Getting started with Laravel	alepeino, Alphonsus, boroboris, Colin Herzog, Community, Ed Rands, Evgeniy Maynagashev, Gaurav, Imam Assidiqqi, James, Ketan Akbari, Kovah, Lance Pioch, Marek Skiba, Martin Bean, Misa Lazovic, nyedidikeke, Oliver Adria, Prakash, rap-2-h, Ru Chern Chong, SeinopSys, Tatranskymedved, Tim
2	Artisan	Alessandro Bassi, Gaurav, Harshal Limaye, Himanshu Raval, Imam Assidiqqi, Kaspars, Laurel, Rubens Mariuzzo, Safoor Safdar, Sagar Naliyapara, SeinopSys
3	Authentication	Aykut CAN, Imam Assidiqqi
4	Authorization	Daniel Verem
5	Blade Templates	A. Raza, agleis, Akshay Khale, alepeino, Alessandro Bassi, Benubird, cbaconnier, Christophvh, Imam Assidiqqi, matiaslauriti, Nauman Zafar, rap-2-h, Safoor Safdar, Tosho Trajanov, yogesh
6	Cashier	littleswany, RamenChef
7	Change default routing behaviour in Laravel 5.2.31 +	Frank Provost
8	Collections	A. Raza, Alessandro Bassi, Alex Harris, bhill77, caoglish, Dummy Code, Gras Double, Ian, Imam Assidiqqi, Josh Rumbut, Karim Geiger, matiaslauriti, Nicklas Kevin Frank, Ozzy, rap-2-h, simonhamp, Vucko
9	Common Issues & Quick Fixes	Nauman Zafar
10	Constants	Mubashar Iqbal, Oscar David, Zakaria Acharki
11	Controllers	Ru Chern Chong
12	Cron basics	A. Raza
13	Cross Domain Request	Imam Assidiqqi, Suraj
14	Custom Helper	Ian, Luceos, rap-2-h, Raunak Gupta

	function	
15	CustomException class in Laravel	ashish bansal
16	Database	A. Raza, adam, caoglish, Ian, Iftikhar uddin, Imam Assidiqqi, liamja, Panagiotis Koursaris, RamenChef, Rubens Mariuzzo, Sanzeeb Aryal, Vucko
17	Database Migrations	Chris, Chris White, Hovsep, hschin, Iftikhar uddin, Imam Assidiqqi, Kaspars, liamja, littleswany, mnoronha, Nauman Zafar, Panagiotis Koursaris, Paulo Freitas, Vucko
18	Database Seeding	Achraf Khouadja, Andrew Nolan, Dan Johnson, Isma, Kyslik, Marco Aurélio Deleu
19	Deploy Laravel 5 App on Shared Hosting on Linux Server	Donkarnash, Gayan, Imam Assidiqqi, Kyslik, PassionInfinite, Pete Houston, rap-2-h, Ru Chern Chong, Stojan Kukrika, ultrasamad
20	Directory Structure	Kaspars, Moppo, RamenChef
21	Eloquent	aimme, alepeino, Alessandro Bassi, Alex Harris, Alfa, Alphonsus, andretzermias, andrewtweber, Andrey Lutskevich, aynber, Buckwheat, Casper Spruit, Dancia, Dipesh Poudel, Ian, Imam Assidiqqi, James, James, jedrzej.kurylo, John Slegers, Josh Rumbut, Kaspars, Ketan Akbari, KuKeC, littleswany, Lykegenes, Maantje, Mahmood, Marco Aurélio Deleu, marcus.ramsden, Marek Skiba, Martin Bean, matiaslauriti, MM2, Nicklas Kevin Frank, Niklas Modess, Nyan Lynn Htut, patricus, Pete Houston, Phroggyy, Prisoner Raju, RamenChef, rap-2-h, Rubens Mariuzzo, Sagar Naliyapara, Samsquanch, Sergio Guillen Mantilla, Tim, tkausl, whoan, Yasin Patel
22	Eloquent : Relationship	Advaith, aimme, Alex Harris, Alphonsus, bhill77, Imam Assidiqqi , Ketan Akbari, Phroggyy, rap-2-h, Ru Chern Chong, Zulfiqar Tariq
23	Eloquent: Accessors & Mutators	Diego Souza, Kyslik
24	Eloquent: Model	Aeolingamenfel, alepeino, Alex Harris, Imam Assidiqqi, John Slegers, Kaspars, littleswany, Marco Aurélio Deleu, marcus.ramsden, Marek Skiba, matiaslauriti, Nicklas Kevin Frank, Samsquanch, Tim
25	Error Handling	Isma, Kyslik, RamenChef, Rubens Mariuzzo

26	Events and Listeners	Bharat Geleda, matiaslauriti, Nauman Zafar
27	Filesystem / Cloud Storage	Imam Assidiqqi, Nitish Kumar, Paulo Laxamana
28	Form Request(s)	Bookeater, Ian, John Roca, Kyslik, RamenChef
29	Getting started with laravel-5.3	A. Raza, Advaith, Community, davejal, Deathstorm, Manish, Matthew Beckman, Robin Dirksen, Shital Jachak
30	Helpers	aimme
31	HTML and Form Builder	alepeino, Casper Spruit, Himanshu Raval, Prakash
32	Installation	A. Raza, alepeino, Alphonsus, Black, boroboris, Gaurav, Imam Assidiqqi, James, Ketan Akbari, Lance Pioch, Marek Skiba, Martin Bean, nyedidikeke, PaladiN, Prakash, rap-2-h, Ru Chern Chong, Sagar Naliyapara, SeinopSys, Tim
33	Installation Guide	Advaith, Amarnasan, aynber, Community, davejal, Dov Benyomin Sohacheski, Imam Assidiqqi, PaladiN, rap-2-h, Ru Chern Chong
34	Introduction to laravel-5.2	A. Raza, ashish bansal, Community, Edward Palen, Ivanka Todorova, Shubhamoy
35	Introduction to laravel-5.3	Ian
36	Laravel Docker	Dov Benyomin Sohacheski
37	Laravel Packages	Casper Spruit, Imam Assidiqqi, Ketan Akbari, rap-2-h, Ru Chern Chong, Tosho Trajanov
38	lumen framework	maksbd19
39	Macros In Eloquent Relationship	Alex Casajuana, Vikash
40	Mail	Yohanan Baruchel
41	Middleware	Alex Harris, Kaspars, Kyslik, Moppo, Pistachio
42	Multiple DB Connections in Laravel	4444, A. Raza, Rana Ghosh
43	Naming Files when uploading with	Donkarnash, RamenChef

	Laravel on Windows	
44	Observer	matiaslauriti, Szenis
45	Pagination	Himanshu Raval, Iftikhar uddin
46	Permissions for storage	A. Raza
47	Policies	Tosho Trajanov
48	Queues	Alessandro Bassi, Kyslik
49	Remove public from URL in laravel	A. Raza, Rana Ghosh, ultrasamad
50	Requests	Ian, Jerodev, RamenChef, Rubens Mariuzzo
51	Route Model Binding	A. Raza, GiuServ, Vikash
52	Routing	A. Raza, alepeino, Alessandro Bassi, Alex Juchem, beznez, Dwight, Ilker Mutlu, Imam Assidiqqi, jedrzej.kurylo, Kyslik, Milan Maharjan, Rubens Mariuzzo, SeinopSys, Vucko
53	Seeding	A. Raza, Alphonsus, Ian, Imam Assidiqqi, Kyslik, SupFrost, whoan
54	Services	A. Raza, El_Matella
55	Socialite	Jonathon, Marco Aurélio Deleu
56	Sparkpost integration with Laravel 5.4	Alvin Chettiar
57	Task Scheduling	Jonathon
58	Testing	Alessandro Bassi, Brayniverse, caoglish, Julian Minde, Kyslik, rap-2-h, Sven
59	Token Mismatch Error in AJAX	Pankaj Makwana
60	use fields aliases in Eloquent	MM2
61	Useful links	Jakub Kratina
62	Valet	David Lartey, Dov Benyomin Sohacheski, Imam Assidiqqi, Misa Lazovic, Ru Chern Chong, Shog9
63	Validation	A. Raza, alepeino, Alessandro Bassi, Alex Harris, Andrew Nolan

		, happyhardik, Himanshu Raval, Ian, Iftikhar uddin, John Slegers, Marco Aurélio Deleu, matiaslauriti, rap-2-h, Rubens Mariuzzo, Safoor Safdar, Sagar Naliyapara, Stephen Leppik, sun, Vucko