**Laravel**

* **MVC** PHP Framework.
* Created by **Taylor Otwell**.
* Helps to develop robust (strong and healthy; vigorous), pragmatic (dealing with things sensibly and realistically in a way that is based on practical rather than theoretical considerations) and more structured application.

Advantage of using Larvel

* More scalable web framework.
* Uses components from other framework to save time in web development.
* Includes namespaces and interfaces, thus helping to organize and manage resources.

Composer

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

It is a tool which includes all the dependencies and libraries. Third party libraries can be installed easily with the help of composer.

All dependencies are noted inside source >> **composer.json**

Artisan (command line interface used in Laravel)

* It includes a set of commands incorporated form **Symphony framework**.

Features of Laravel

* **Modularity:** Laravel provide 20 built in Libraries and modules. Every module is instigated with composer dependency manager which eases updates.
* **Testability:** Laravel includes features and helpers which helps in testing through various test cases.
* **Routing:** Provides a flexible approach to the programmer to define routes in web application. Helps to scale the application in better way and increases performance.
* **Configuration Management:** Provides a consistence approach to handle the configuration in an efficient way. Change in configuration is needed to run the application to run on different environments.
* **Query Builder and ORM:** Laravel incorporates a query builder which helps in querying Database using various chain methods. It provides **ORM (Object Relational Mapper)** and **ActiveRecord** implementation call **Eloquent**.
* **Schema Builder:** It maintains the database definition and schema in PHP code. It also maintains a track of changes with respect to database migration.
* **Templet Engine:** Laravel user **Blade Template** engine, a lightweight template language used to design hierarchical blocks and layouts with predefined blocks that include dynamic contents.
* **E-mail:**  Laravel includes a mail class which helps in sending mail with rich content and attachments from the web application.
* **Authentication:** Laravel eases designing authentication as it includes features such as **register**, **forgot password**, and **send password reminders**.
* **Redis:** Laravel uses **Redis** to connect to existing sessions and general-purpose-cache. **Redis** interacts with sessions directly.
* **Queues:** Laravel includes queue services like emailing large number of users or a specified **Cron** job. These queues help in completing tasks in an easier manner without waiting for the previous task to be completed.
* **Event and Common Bus:** Laravel 5.1 includes **Command Bus** which helps in executing commands and dispatch events in a simple way. The commands in Laravel act as per the application’s lifecycle.

# **Laravel – Installation**

**Step 1** − Download composer to install. [https://getcomposer.org/download/](https://getcomposer.org/download/" \t "_blank)

**Step 2** − Check the installation by typing the Composer command in the command prompt. C:\>composer

**Step 3** − Create a new directory anywhere in your system for your new Laravel project. After that, move to path where you have created the new directory and type the following command there to install Laravel.

#### Via Laravel Installer

>composer global require laravel/installer

>laravel new blog

#### Via Composer Create-Project

>composer create-project --prefer-dist laravel/laravel blog

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

**Step 4** − The above command will install Laravel in the current directory. Start the Laravel service by executing the following command.

php artisan serve

**Making New Controller in Laravel:-**

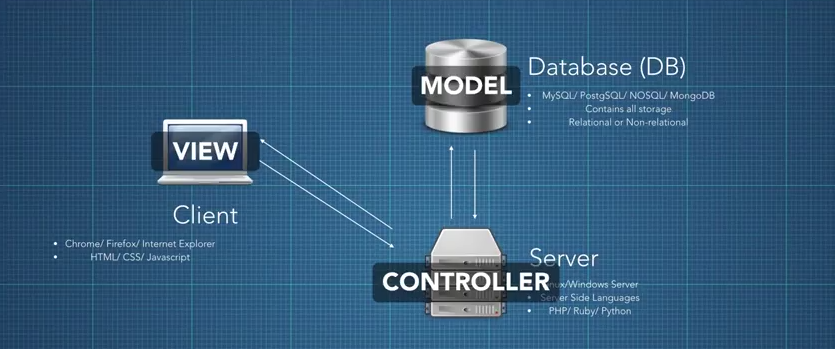
>php artisan make:controller <controller\_name>

1. Controller -> Logic

Controller is the place where logic happens.

1. Model -> Database Operations
2. Views -> Front end Display(HTML & Such)

MVC:

******MVC is an architectural paradigm for programming designed in 1979.

1. The Model (Interacting with your data)

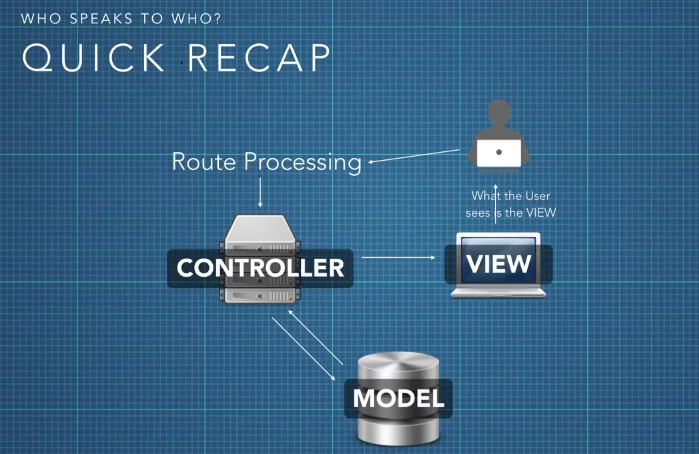
* Adding and retrieving data from the DB.
* Processing the data from or to the database.
* Speaks only with the controller.

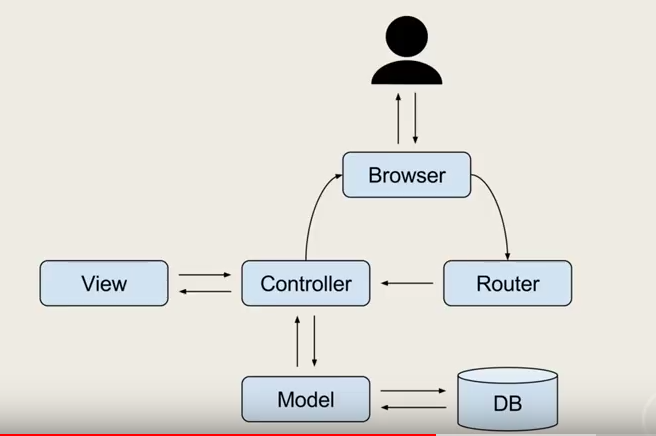
1. The View (Displaying to the user)

* This is the only thing the user ever see.
* Think of good old-fashioned HTML/CSS.
* Listen only to the controller.

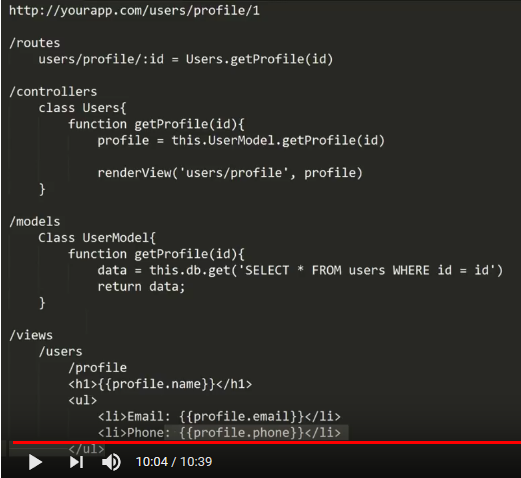
1. The Controller (Interacting with the user)

* Processes GET/POST/PUT/DESTROY requests.
* All server side-logic.
* The Middle Man
  + Takes Info from user.
  + Process info and takes it to DB if needed.
  + Receives info from the DB
  + Speaks to the view to explain presentation to the viewer.



****

**A sudo example to understand MVC:-**



**Routing to View in Laravel : -**

Route::get('/', function () {

return view('welcome');

});

**Routing to View via controller in Laravel : -**

Route::get('/about', 'PagesController@getAbout');

# **Passing Data to Views in Laravel**

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

return 'user has an id ='.$id;

});

Blade Templet Engine in Laravel

<?php echo $usernamne;?> can be written as {{$username}}

**Passing Dynamic values to the Blade templet engine using an associate array in Laravel**

Route::get('/comment/{username}/{id}', function ($username,$id) {

return view('welcome',['username'=>$username,'id'=>$id]);

});

//or

**Passing Dynamic values to the Blade templet engine using with() in Laravel**

Route::get('/comment/{username}/{id}', function ($username,$id) {

return view('welcome')->with('username',$username)->with('id',$id);

});

return view('pages.about')->with("fullname", $full);

OR

return view('pages.about')->withFullname($full);

return view('pages.about')->withFullname($full)->withEmail($email);

in view pages access using: -

<div class="title m-b-md">

What do you know about {{$fullname}} ?

</div>

<p>Email me @ {{$email}}</p>

**Master Pages:-**

@yield('<content here>') - - Give us the content we want.

Eg;

@yield(‘title’)

@yield(‘body’)

**Extending a Master Page to a child page: -**

**@**extends(‘<path of master page>’)

Eg;

*@extends(‘layouts.app’)* -> extending from the app.blade.php inside the layout folder.

**Providing the yield contents to Master Page from the child page:-**

@section(‘<yield content>’)

Eg;

@section(‘title’,’Home’)

**Placing child page content inside the section**

@section(‘<name of section>’)

<section body>

@endsection

Eg; -

@section(‘body’)

<Place the section body here>

@endsection

Blade Conditional STATEMENTS: -

**Passing the parameters for the conditional statements of blade templating engine:**

Route::get('home2', function () {

$processors = ['Intel i7', 'Intel i5', 'Intel i3', 'AMD A10'];

$num1 = 45;

$num2 = 23;

return view('home2')->with('processors', $processors)->with('num1',$num1)->with('num2',$num2);

});

**foreach loop for blade templating engine**

@foreach($processors as $processor)

{{$processor}}<br>

@endforeach

**if else condition in blade templating engine**

**@if($num1>$num2)**

**Num1 is Greater than Num2 whose value is {{$num1}}**

**@else**

**Num2 value exceeds Num1 value whose value is {{$num2}}**

**@endif**

**while condition in blade templating engine**

@while($num2<50)

{{$num2++}}

@endwhile

**unless condition in blade templating engine(means if !)**

@unless($num1>$num2) // way of telling if !

Num2 is Greater than Num1 whose value is {{$num2}}

@else

Num1 is Greater than Num2 whose value is {{$num1}}

@endunless

**for loop in blade templating engine**

@for ($i = 0; $i < 10; $i++)

The current value is {{ $i }}

@endfor

Laravel Migrations

## It is a way to create table in MySQL by providing commands inside a Laravel file (database->migrations) and its way to perform version control so we can rollback to whatever version we have.

We will create a table without getting into the MySQL interface at all.

# Steps involved in creating a migration file: -

1. Creating new migration file: -

php artisan make:migration create\_<table name>\_table --create=<table name>

**eg;** To create a table “articles”

php artisan make:migration create\_articles\_table --create=articles

1. Add the new fields in newly created migration file(database>migration): -

public function up()

{

Schema::create('articles', function (Blueprint $table) {

**$table->increments('id');**

**$table->string('name');**

**$table->text('content');**

$table->timestamps();

});

}

1. Perform the migration

>php artisan migrate

1. Rolling back migrations

>php artisan migrate:rollback

1. For help in migrations

>php artisan help

>php artisan help make:migration

1. For making any changes in the present tables

>php artisan make:migration add\_categories\_to\_artical\_table –table=articals

1. Add the new fields in newly created migration file: -

public function up()

{

Schema::table('articals', function (Blueprint $table) {

**$table->string('categories');**

});

}

1. Perform the migration

>php artisan migrate

1. To completely change the table structure (refresh drop all your tables and delete all your data and rebuild all tables for you)
2. make change on the file

public function up()

{

Schema::create('articals', function (Blueprint $table) {

$table->increments('id');

$table->string('name');

**$table->string('content\_title');**

**$table->text('content\_body');**

$table->timestamps();

});

}

1. give the refresh command

>php artisan migrate:refresh

Eloquent ORM (Object Relational Mapping)

With eloquent we are converting everything to be used as a class.

# Making model using artisan: -

>php artisan make:model <Model Name>

Model make everything to be used as a class.

Naming convention for model:

1. Take the name of table whichever you making model for and make it singular noun.
2. The first letter needs to be Uppercase.

Creating a model for “**articles**” table:-

>php artisan make:model **Article**

It will Create a Model “Article inside” app>Article.php

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model

{

}

1.We create view pages inside resources>views>articles>create.blade.php

2.We created a Route for create article page inside Route>web.php

Route::get('/articles/create', 'ArticlesController@create');

3.Create controller ArticlesController by using artisan command

>php artisan make:controller ArticlesController

4. New controller Ready inside app>Http>Controllers>ArticlesController.php

5.Create a form to enter Articles detail inside create.blade.php

6.Create a function inside the ArticlesController.php to return create.blade.php view

public function create(){

return view('articles.create');

}

7.Give the path to create.blade.php from to submit the data to by using

<form action="/articles/submit" method="post">

Also mention the cref\_token

<input name="\_token" type="hidden" value="{{ csrf\_token() }}"/>

8.Create the route for the submit path and assign the controller for submit

Route::post('/articles/submit', 'ArticlesController@submit');

9.Inside the controller ArticlesController.php create a function for the submit action (Eloquent is used here to submit the date to the database)

//Gaurav

use App\Article; //This is needed to make the Article object.

public function submit(Request $request){ // $request will contain all the data submitted through //the post request of form

// return $request->all();//to print the all the submitted data in JSON form

$article = new Article;

$article->name = $request->name;

$article->content\_title = $request->content\_title;

$article->content\_body = $request->content\_body;

$article->categories = $request->categories;

$article->save();

}

\* [Rakshith Vasudev](https://www.youtube.com/channel/UCCfS00z2jVV4tS3xqsGjvrw) skipped Laravel Collective because he is not a huge fan of that.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Laravel blog project

**Process followed inside a controller**

1. Process variable data or parameters.
2. Talk to the model.
3. Receive from the model.
4. Compile or process data from the model if needed.
5. Pass the data to the correct view.

**Layout in Blade: -**

Used to manage the reparative elements of all the pages into one single place.

**Main Page**

Put @yield(‘content’) inside the main page where customized content is needed

**Child Pages**

Extent the main page first @extends(‘main’)

Enclose the child page inside the @section(‘content’)

Eg;

@section(‘content’)

<customized page content here>

@endsection

**Partials with Blade: -**

Partials are used to make our code clean, we can put our coding section in a different files and folders and include them into our file

Eg;

@include(‘<path of file here>’)

\*Naming convention for partials is that they start with an underscore (\_)

Eg.

<head>

@include('partials.\_head')

</head>

And we have partial containing head section into the directory

*partials > \_head.blade.php*

# **Models**

Model naming convention (inside app folder)

1. Capitalize the first letter.
2. Use singular noun.

\*Usually we have 1 model per table (There are exceptions).

For e.g.; For **posts** table we will have a model name **Post** (Singular noun and Capital first latter)

Create model and migration for table using the same command

* *php artisan make:model Post - -migration*

*OR*

* *php artisan make:model Post -m*

*(- -migration OR -m flag is used to create a migration with the model)*

# Migrations:

These are the step by step instructions that allow Laravel to recreate database structures and database tables without us doing it in the phpMyAdmin.

Migration command: - >php artisan migrate

We follow CRUD operations in any Application:

CREATE READ UPDATE DELETE

Controllers

\*Naming convention to name the controller is Controller name in singular noun + Controller in camel casing

E.g.; **>php artisan make:controller PostController - -resource**

(it’s used to make a crud controller)

**(-- resource** flag is used to get the basic CRUD structure inside our newly created controller by default)

CRUD operation functions created inside the controller PostController by

- -resource flag: -

1. index() - Display the listing of the resource
2. create() - Show the form for creating a new resource.
3. store(Request $request) - Store a newly created resource in storage.
4. show($id) - Display the specified resource.
5. edit($id) - Show the form for editing the specified resource.
6. update (Request $request, $id) - Update the specified resource in storage.
7. destroy($id) - Remove the specified resource from storage.

**Creating Route for all the CRUD functions above that are created using the our –resource flag with controller -**

**Route::resource('posts', 'PostController');**

(by telling Laravel we are managing the resources it knows to expect all that commands that are already in our controller)

\*To get the list of all the routs in our application

>php artisan route:list

Curd operations: -

1. Creating a from

public function create()

{

return view('posts.create');

}

Laravel Collective

\*Removed form the core Laravel 5.

Helps to create form using Laravel

Go to [laravelcollective.com](https://laravelcollective.com/docs/master/html) > Install Laravel collective.

>composer require "laravelcollective/html":"^5.4.0"

Next, add your new provider to the providers array of config/app.php:

'providers' => [

// ...

Collective\Html\HtmlServiceProvider::class,

// ...

],

Finally, add two class aliases(Facade) to the aliases array of config/app.php:

'aliases' => [

// ...

'Form' => Collective\Html\FormFacade::class,

'Html' => Collective\Html\HtmlFacade::class,

// ...

],

Opening A Form:-

{!! Form::open(['url' => 'foo/bar']) !!}

//

{!! Form::close() !!}

Named Route(used to manage routes)

\*Check the Name filed for name route in >php artisan route:list

To use named routes change the following in form

‘url’ => ‘foo/bar’ 🡪 'route' => 'posts.store'

Form Elements:-

**Form fields –**

****{{ Form:label(‘<column\_name\_of\_DB>’, ‘label\_name’) }}****

**{{ Form:<field\_type>(‘<field name as in the DB table>’, ’<default value>’, ’<array\_of \_any\_other\_options\_we\_have>’)}}**

{{ Form::label('title','Title:') }}

{{ Form::text('title', null, ['class' => 'form-control']) }}

{{ Form::label('body', 'Post Body:') }}

{{ Form::textarea('body',null, ['class' => 'form-control']) }}

{{ Form::submit('Create Post', ['class' => 'btn btn-success btn-lg btn-block', 'style' => 'margin-top:20px;']) }}

1. Storing the form :

public function store(Request $request)

{

**//Validate the data using laravel validation**

$this->validate($request, array(

'title' => 'required|max:255',

'body' => 'required'

));

**//Store in the database(Using Eloquent)**

$post = new Post;

$post->title = $request->title;

$post->body = $request->body;

$post->save();

**// Redirect to another page(Redirecting to show the using Named Routes)**

return redirect()->route(‘post.show’, $post->id); **//getting the id of the newly created post using the post object**

}

\*If the validation fails, we will be rendered back to the create action

\* **use App\Post**; to use the Post model class

Form Validation using JavaScript Library – Parsley

Download Parsley js from <http://parsleyjs.org/doc/download.html> and save it directly on your Laravel project >public>js>parcley.min.js

Download Parsley CSS and save it under your Laravel project > public>css>parcley.css (we can create our own CSS too)

Use the CSS and js using the **@section** into the page we want them to be. We have already created **@yield(‘javascript’)** in **main.blade.php** and **@yield('stylesheet')** inside the **partials.\_head.blade.php** for customized javascript and css.

On create.blade.php add the section for custom JavaScript

# @section('script')

{{!! Html::script('js/parsley.min.js') !!}}

@endsection

-giving the path to the CSS file inside our public folder using the raw code for the **html helpers** we installed in collective while installing it for **form helpers**

On create.blade.php add the section for custom stylesheet

@section('stylesheets')

{{!! Html::style('css/parsley.css') !!}}

@endsection

Used parsley html validation to validate the form fields

{!! Form::open(['route' => 'posts.store', **'data-parsley-validate' => ''**]) !!}

{{ Form::label('title','Title:') }}

{{ Form::text('title', null, ['class' => 'form-control'**, 'required' => '', 'maxlength' => '254'])** }}

{{ Form::label('body', 'Post Body:') }}

{{ Form::textarea('body',null, ['class' => 'form-control', **'required' => ''**]) }}

{{ Form::submit('Create Post', ['class' => 'btn btn-success btn-lg btn-block', 'style' => 'margin-top:20px;']) }}

{!! Form::close() !!}

Sessions in Laravel:

\*Laravel default session time is 120 minutes.

**Setting Session for project:-**

config>Session.php ----used to edit the settings for the session

Flash Session:-

A special kind of session that only exists over the next one request and get deleted after. (They are used basically for massages)

**SYNTAX: -**

**Session::flash('key', 'value');**

eg; Used to set message in PostController store method after the save() when post get saved into the database-

**Session::flash('success', 'The blog post was successfully saved!');**

To make the session save permanently use put() (until session get expired)

**Session::put('key', 'value');**

eg; Session::put(‘success’,’The blog post was successfully saved!’);

**“flash” exists for one-page request**

**“put” exists until the session is removed**

\*write “use Session;” namespace to use session class.

**Setting the show() inside PostController to display the newly created post with the success message saved in the session –**

public function show($id)

{

return view('posts.show');

}

create file -> resources>views>posts> show.blade.php.

**Displaying the success massage stored in the session: -**

Inside resources>views>partials>\_message.blade.php

@if(Session::has('success'))

<div class="alert alert-success" role="alert">

<strong>Success:</strong>{{ Session::get('success') }}

</div>

@endif

Handling error messages in Laravel:

In the PostController inside the store() the $this->validate() will automatically summits the error back to the same request i.e. the previous page i.e. the create() post.create page.

//Validate the data

$this->validate($request, array(

'title' => 'required|max:255',

'body' => 'required'

));

It automatically adds a message called “errors” to the flash session, it automatically flash it to the session like we did manually did with the success message. “**errors**” is an object includes multiple errors inside

There always going to be some errors set inside the error object by Laravel (zero items in the object) default so it always returns true.

Inside resources>views>partials>\_message.blade.php

@if(count($errors) > 0)

<div class="alert alert-danger" role="alert">

<strong>Errors:</strong>

<ul>

@foreach($errors->all() as $error)

<li>{{ $error }}</li>

@endforeach

</ul>

</div>

@endif

* **Put your js and css inside the js and css folders inside the public folder of the project directory.**
* **Everything that is inside the public folder is accessible to the world through the internet.**
* **Helpers for css and js**
* **styles(‘css/styles.css’) and scripts(‘js/javascripts.js’)**
* **To list all the routes we have >php artisan route:list**

1. **Reading the data, reading post (post.show)**

Inside the PostController **show($id)** we get the id of the post passed by the route and finds the post by the id no. through the “Post” model and calls the view to show the post data: -

Eloquent find() is used to find the item by primary ID

***$post = Post::find($id);***

It will extract all the information about that id and we can pass it to our view to display the post data.

*return view(‘posts.show’)->with(‘post’, $post);*

*or we can use shortcut*

*return view(‘posts.show’)->withPost($post);*

***\****

***{{}} ----- echo out the data***

***{!! !!} ------ do not echo out the data***

In the view show.blade.php we will display the content of the post that we passed through the $post

<h1> {{$post->title}} </h1>

<p class="lead">{{$post->body}}</p>

It is time to work on the R from CRUD, or “Read”. Now we focus on the posts.show function of our controller. This function is designed to show a specific blog post to the user. The url for posts.show is /posts/{posts} or in other words the {posts} will represent the primary id number for the post that we want to view. So, the user’s URL will look like this /posts/1 or /posts/124 and this will signify to Laravel that we want to view the post with the primary id of 1 or 124 (depending on which URL we used above). Our controller is simple, we will create a variable called $post and then we will search the database by the primary id that was passed into our project. We can do this with this simple line of code: **$post = Post::find($id);** The find() method is a special method our models have that allow us to find a resource by primary id. This is the fastest way to search the database, the primary id column is indexed and Laravel will stop at the first one, making our queries much faster than usual. We make sure to pass our variable into the view so we can access it in the view. From the view we can call values based on the name of the columns. Use blade to output the value of the variables.

we also display the post dates inside show.blade.php :

<dl class="dl-horizontal">

<dt>Created At:</dt>

<dd>{{date('M j, Y h:ia', strtotime($post->created\_at))}}</dd>

</dl>

<dl class="dl-horizontal">

<dt>Last Updated:</dt>

<dd>{{date('M j, Y h:ia', strtotime($post->updated\_at))}}</dd>

</dl>

Link Route:

It is a html helper, part of Laravel collective. It allows us to pass a named route and get a full anchor tag.

Html::linkRoute() which allows us to give it a named route and it generates a whole html anchor tag (aka link) , it’s a more Laravel way of making a link.

The first argument is the named route (string), the second is the value of the anchor tag(display name), the third is an array (empty if no url params, or add the params in order) we are passing the id of the post to edit, and the final argument is another array of all the options you want to use e.g., the bootstrap class.

<div class="row">

<div class="col-sm-6">

{!! Html::linkRoute('posts.edit', 'Edit', array($post->id), array('class' => 'btn btn-primary btn-block') ) !!}

<!--<a href="" class="btn btn-primary btn-block">Edit</a>-->

</div>

<div class="col-sm-6">

{!! Html::linkRoute('posts.destroy', 'Delete', array($post->id), array('class' => 'btn btn-danger btn-block') ) !!}

<!--<a href="" class="btn btn-danger btn-block">Delete</a> ---------->

</div>

</div>

**Reading all the data of post and displaying it using the index() of PostController:**

1. create a variable in the index() of PostController and store all the blog post data in it form the database

$posts = Post::all();

Or

$posts = Post::orderBy('id', 'desc')->paginate(5);

We us eloquent all() to extract the data of all the posts from the posts table of database.

1. return a view from PostCotroller and pass in the above variable

return view('posts.index')->with('posts', $posts);

//or

return view('posts.index')->withPosts($posts);

1. create a view index.blade.php inside the posts folder of view

\*route helper allows us to pass in a named route and then it will automatically generate the URL from that name route.

<a href="{{ route('posts.create') }}" class="btn btn-lg btn-primary btn-h1-spacing">Create New Post</a>

Here the route() will link us to create a new blog post

\*Using Html helper to mention css styling path:-

{{ Html::style('css/style.css') }}

It will automatically create the external css link <link rel=”stylesheet” href=””>

@foreach($posts as $post)

<tr>

<th>{{$post->id}}</th>

<td>{{$post->title}}</td>

<td>{{substr($post->body, 0, 50)}}{{ strlen($post->body)>50?"...":"" }}</td>

<td>{{date('j/M/Y',strtotime($post->created\_at))}}</td>

<td>

<a href="{{route('posts.show',$post->id)}}" class="btn btn-sm btn-primary">View</a>

<a href="{{route('posts.edit', $post->id)}}" class="btn btn-sm btn-info">Edit</a></td>

</tr>

@endforeach

\*using foreach loop to print all the posts data.

\*Using conditional statement (ternary operator) to print “…” if the no of character of body is greater than 50

\*using route to give the route to the anchor tags.

\*To highlight the tab which is active we make the corresponding navbar class active inside the partial \_nav.blade.php.

Request is a class of helper file in Laravel, we use it to check if we are on a current page

<li class="nav-item {{ Request::is('/') ? "active" : "" }}">

**CRUD**

**C =**

1. create() – opening up a form to create a new post, GET request
2. store(Request $request) – Storing the post into the database. POST request

**U =**

1. edit($id) – show a from with the data filled into that form. GET request
2. update(Request $reqest, $id) – submits the data in to the database and update the db table using the POST request.

**R =**

1. show($id) – Show the individual post.
2. index() – Show all posts in the database table

Editing and Updataing the Post data:

In the PostController under the edit($id) function.

### public function edit($id)

### {

### //find the post in the database and save it as a variable.

### $post = Post::find($id);

### //return the view and passing that variable

### return view('posts.edit')->with('post', $post);

### }

Model Form Binding:

Here we can pass in a model object, like we did in our controller we create a model object $post = **Post**::find($id); which finds the model for the database and stores it in the variable $post and then we pass the object into the view.

\*We should have Laravel collective form binders (form helpers) to use this functionality. see part 11 of video series for Laravel collective installation.

{!! Form::model($post, ['route'=>['posts.update', $post->id] , 'method'=>'PUT']) !!}

{{ Form::label('title', 'Title:')}}

{{ Form::text('title', null, ['class'=>'form-control input-lg']) }}

{{ Form::label('body', 'Body:', ['class'=>'form-spacing-top'])}}

{{ Form::textarea('body', null, ['class'=>'form-control']) }}

It tells Laravel that we are opening a form but we need to connect it to a model that we’re passing in.

When we make our view, it’s time to learn about Model/Form binding. This is when we take a form and "bind" it (connect it) to our model. Since we passed in a model object into our view, we can use form helpers to bind the model object into our form. This means that our form will automatically fill in the data that is already in our database, making it super easy to then make our changes and submit them with the save button.

**Updating the post data:**

public function update(Request $request, $id)

{

//Validate the data

$this->validate($request, array(

'title' => 'required|max:255',

'body' => 'required'

));

//Save the data to the database

$post = Post::find($id);

$post->title = $request->title;

//or

//$post->title = $request->input('title');

$post->body = $request->body;

//or

// $post->body = $request->input('body');

$post->save();

//set flash data with success message

Session::flash('success', 'The blog post was updated successfully saved!');

//Redirect whth flash data to posts.show

return redirect()->route('posts.show', $post->id);

}

**Deleting the post data:**

{!! Form::open(['route' => ['posts.destroy', $post->id], 'method' => 'DELETE']) !!}

{!! Form::submit('Delete', ['class' => 'btn btn-danger btn-block']) !!}

{!! Form::close() !!}

Create a small form with a button in show.blade.php

**Post controller delete method.:-**

public function destroy($id)

{

$post = Post::find($id);

$post->delete();

Session::flash('success','The post was successfully deleted.');

return redirect()->route('posts.index');

}

The first thing to remember is that you need to protect this route. We wouldn't want to make the delete available as a basic "GET" request to a URL like "localhost:8000/posts/1/delete" because this could allow people to accidentally delete posts. So, what we want to do is protect it a bit and require them to use certain methods and forms. That is why the delete functions need to be protected under the HTTP Method: DELETE. This can’t be done by normal users outside of the forms we define.

Pagination in Laravel:

Pagination is the simple links at the bottom of the page that help us navigate large amounts of data. Pagination in the Laravel Framework is easy. We can call the simple paginate() method in PostController and Laravel does the rest.

$posts = Post::paginate(5);

It fetches a result for 5 posts only.

we can show the page with the pagination links to move between the pages next and previous. we put the following code after the table in the view>posts>index.blade.php

<div class="text-center">

{!! $posts->links(); !!}

</div>

to order our results in descending order, to show the most recent items on top. This makes it easier to manage the posts we are working on now, which are the most recent posts.

Post::orderBy('id', 'desc');

$posts = Post::orderBy('id', 'desc')->paginate(5); // in PostControllers index()

Query Builder in Laravel: Eloquent (ORM)

\*look more into the documentation.

It allows us to do more complex queries.

To get the recent 4 posts on the welcome page we use query builder inside the getIndex() of pageController.

$posts = Post::orderBy('created\_at','desc')->limit(4)->get();

Passing this $post variable containing the recent 4 posts to the view pages.welcome-

return view('pages.welcome')->withPosts($posts);

Inside the welcome.blade.php we print the recent 4 posts using the foreach loop

@foreach($posts as $post)

<div class="post">

<h3>{{ $post->title }}</h3>

<p>{{ substr($post->body, 0, 80) }}{{ strlen($post->body) > 80 ? "...": ""}}</p>

<a href="#" class="btn btn-primary">Read More</a>

</div>

<hr>

@endforeach

**Link Route Button code to see all the posts: - << See All Posts**

We used html helper to create a link route and pass the name of the route post.index and then the value we want to show the third item on the link route is for any parameters that we want to use and last we pass in the classes.

<div class="col-sm-12">

{{ Html::linkRoute('posts.index', '<< See All Posts', [], ['class' => 'btn btn-default btn-block btn-h1-spacing']) }}

</div>

Adding slug URLs to our Blog Post:

1. Creating migration

php artisan make:migration add\_slug\_to\_posts

1. Create the up and down migration functions

public function up()

{

Schema:table('posts', function($table) {

$table->string('slug')->unique()->after('body');

});

}

public function down()

{

Schema:table('posts', function($table){

$table->dropColumn('slug');

});

}

//for indexing (for faster db searching ) we are making the slug column as

indexed column i.e. unique

1. To use dropColumn() add

"doctrine/dbal" : "\*" into the composer.json

1. Perform the migration :--

php artisan migrate --- migrate a change to the database

php artisan migrate:rollback ----rollback the recent migration

php artisan migrate:reset --- rollback all the migration at once

**php artisan migrate:refresh** ---rollback all the migration at once and then re-migrate them again to the database.

1. Create a form field for the slug in the create.blade.php inside the views>posts

{{ Form::label('slug', 'Slug:')}}

{{ Form::text('slug', null, ['class' => 'form-control', 'required' => '', 'minlength' => '5', 'maxlength' => '255'])}}

1. set the validation rule for slug inside the PostController store(). ensuring slug field as unique inside the posts table slug column.

'slug' => 'required|alpha\_dash|min:5|max:255|unique:posts,slug',

1. Store the slug value in the $post object inside the store() of postController

$post->slug = $request->slug;

1. Create the form filed for the slug in the edit.blade.php inside the view>posts

{{ Form::label('slug', 'Slug:')}}

{{ Form::text('slug', null, ['class'=>'form-control']) }}

1. set the validation rule for slug inside the PostController update()

'slug' => 'required|alpha\_dash|min:5|max:255|unique:posts,slug',

1. Update the slug value in the $post object inside the update() of postController

$post->slug = $request->input('slug');

Slugs in our URL Routes:

***Searching the post through the slugs into our URL.***

We want all the pages at the /blog extension e.g.,

*domain.com/blog/slug-goes-here*

1. Set the route for slug operation -

Route::get('blog/{slug}', ['as' => 'blog.single', 'uses' => 'BlogController@getSingle'])->where('slug', '[\w\d\-\\_]+');

we are doing the GET request to the URL with **/blog** in it. **{slug}** is the name of the parameter we use to search the indexed slug column in the DB.

Now we are going to name the route so we can access it really quick with our routes file makes it lot easier to work with. To name the route we need second parameter here **[‘as’=>’blog.single’]**

We are going to call it **blog.single** to represent a single post in the blog.

We need to now add in action so we have somewhere for it to go, so we will add second parameter to our array here.

**[‘uses’=>’BlogController@getSingle’]**

we specify the controller it have to go and the method to execute inside that controller. **get** represent it’s a get request and Single says it’s a single blog post

We are defining what type of information might come in through the slug, we want to basically tell Laravel that if it falls outside these parameters that we don’t want to accept it as a part of this route, because it not going to found in the database it’s not going to be valid and, in that case, it’s just going to return a 404 error. After the **get()** we add **get()->where();** we pass regular expression inside the where(). we allow any words, any numbers, dash and underscore and it restricting it to only those characters anything outside those characters would be rejected.

1. Create a new controller BlogController –

php artisan make:controller BlogController

1. Create the function getSingle($slug) inside the BlogController

class BlogController extends Controller

{

public function getSingle($slug) {

return $slug;

}

}

We are passing a parameter here by route called slug, we are calling it $slug to correspond it to the {slug} parameter we are passing here from route.

by going to <http://localhost:8000/blog/first-post> we will get the name of our slug, because that’s what we are returning here from our getSingle() controller function.

we are sure our controller function grabbing our slug form the URL.

Now we can use it to search our database and find the correct post and then we going to pull that our from the DB and display that post on the screen through view.

1. Fetch from the DB based on slug

$post = Post::where('slug', '=', $slug)->first();

Because we are actually grabbing only one unique item we use first() instead of get(). If we find one item i.e. first item that matches our slug variable we done, and stop searching the DB. get() will give us and object with array so we have to loop through the object to get the item, using first() we are getting the item directly.

1. Return the view and pass in the post object

return view('blog.single')->with('post', $post);

1. We now create a view under views>blog>single.blade.php

@extends('main')

@section('title', “| $post->title”)

@section('content')

<div class="row">

<div class="col-md-8 col-md-offset-2">

**<h1>{{ $post->title }}</h1>**

**<p>{{ $post->body }}</p>**

</div>

</div>

@stop

* Double quotation in php and Laravel do Interpolation i.e. its fetches the value of a variable inside them.
* Single quotation in php and Laravel doesn’t do interpolation and does not put the value of a variable inside it

eg;

**@section('title', "| $post->title")** will get us value of $post->title

**@section('title', ‘| $post->title’)** will give us $post->title as it is but no value

1. Give URL for the Read More Link in the welcome.blade.php

<a href="/blog/{{$post->slug}}" class="btn btn-primary">Read More</a>

-me working

OR

<a href="{{ url(‘blog/’.$post->slug)}}" class="btn btn-primary">Read More</a>

{{url('blog/'.$post->slug)}}

---J Courtis

1. Give URL for the URL link in show.blade.php

\*we are using route method here instead of url method, we can use both

<dl class="dl-horizontal">

<label>URL:</label>

<p>

<a href="{{ route('blog.single', $post->slug) }}">{{ route('blog.single', $post->slug) }}</a>

</p>

</dl>

Solving The “slug already been taken” error while editing the posts: -

Changing the update method of PostController: -

**//Validate the data**

**$post = Post::find($id);**

**if($request->input('slug') == $post->slug){**

**$this->validate($request, array(**

**'title' => 'required|max:255',**

**'body' => 'required'**

**));**

**} else{**

**$this->validate($request, array(**

**'title' => 'required|max:255',**

**'slug' => 'required|alpha\_dash|min:5|max:255|unique:posts,slug',**

**'body' => 'required'**

**));**

**}**

# **Adding Features to our BlogController:-**

Creating the Index Page for blogs(The page that will show the all blog posts):-

1. Create the route from blog index page

Route::get('blog',['uses' => 'BlogController@getIndex', 'as' => 'blog.index']);

We are using the **get request** for **/blog** url, second parameter is an array for all other options, it uses the BlogControllers getIndex method(action) and is named as blog.index(Named Route).

1. (\* method inside a controller are called action) Create our getIndex() action inside BlogController

public function getIndex(){

$posts = Post::paginate(10);

return view('blog.index')->with('post', $posts);

}

1. Creating view index.blade.php inside the blog folder.

\*\*Post Model

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

//

}

Our Post model is extending the Model class which is located at the “Illuminate\Database\Eloquent\Model” path ie, vendor>Laravel>framework>src>Illuminate>Database>Eloquent>Model

# **Laravel Authentication Routes Views**

**>php artisan make:auth**

**Command to create all the authentication routes in Laravel v5.5**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**old method**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Routes: By default Laravel does not set up routes for our authentication system. We need to do that ourselves. We need to set up two routes for logging in (post and get) and then we need to set up a get request for logging out. Finally a get and post request for registering a new user.

You will want to copy the code exactly as put in the code snippets section below.

Views: We need to create only two views for our users. One is a form for logging in and the other is a form for registering a new user. You can really create any type of form you want, but make sure you follow the basic outline:

Form

CSRF Protection (automagically added when you use form helpers).

Email Field

Password Field

Remember Me Checkbox (will work without it, but won't be able to trigger remember functionality)

End Form

1. **Set Authentication Routes: -**

// Authentication

Routes Route::get('auth/login', 'Auth\AuthController@getLogin');

Route::post('auth/login', 'Auth\AuthController@postLogin');

Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Registration Routes

Route::get('auth/register', 'Auth\AuthController@getRegister');

Route::post('auth/register', 'Auth\AuthController@postRegister');

1. **Creating Views: -**
2. create a folder inside the view called “auth”
3. create login.blade.php and register.blade.php(Laravel is expecting the same file names)

**\*we need to add cerf\_field() if we are creating the form with form helpers**

**<form>**

**{!! csrf\_field() !!}**

**</form>**

* **laravel-recipes.com 🡪for Laravel collective help.**

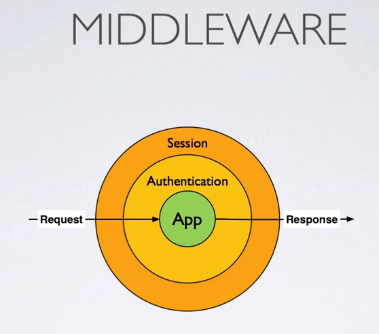
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_end old routes method\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Checking if the user is logged in or logged out in every page by inserting the following code in main.blade.php

<!--returns true or false if a user is logged in or logged out-->

{{ Auth::check() ? "Logged In" : "Logged Out"}}

Middleware



Middleware is the kind of thing that surrounds our application, we pass through the middle ware the way in and way out our Application (Model, View and Controller), it is all around of our application.

Middleware comes in middle of our App and Request/Response

Session and Authentication are Laravel’s built in middleware.

e.g.

public function \_\_construct()

{

$this->middleware('guest');

}

This middleware is called inside the constructor method of login or register controllers.

guest middleware checks whether we are logged in or not. It used to authenticate that we are not logged in before we access the login page or register page, restricting us to first logout before logging or registration as a new user.

public function \_\_construct()

{

$this->middleware('auth');

}

auth middleware is used to check where we are an authenticated user or not i.e. logged in user is allowed to visit

**\*PostController should be accessible only to the logged in user.**

**\*BlogController should be accessible to the guest user as well.**

**Locking PostController so that only the logged in user can access it-:**

At the top of PostController class write the constructor function

public function \_construct(){

$this->middleware('auth');

}

using the auth middleware only the authenticated(logged in user) are able to access the PostController.

**Auth::check()** --- returns true if someone is logged in and false if logged out.

**Auth::guest()** --- returns true if someone is logged out and false if logged in.

**Auth::user()** --- gets the information of the user out of the DB(get logged in user object). Returns an eloquent object.

**Auth::id()** --- similar to user() but only Allows to just get the id number

**public function attempt(array $credentials = [], $remember = false, $login = true)**

* Attempt to log in user, Attempt to authenticate a user using the given credentials.

**public function once(array $credentials = []);**

**--**allows us to log someone in for just one request, Log a user into the application without session or cookies.

**public function loginUsingId($id, $remember = false);**

**--**log user in via user id

**public function onceUsingId($id);**

**--**Log user in for one request via user id, Log the given user ID into the application without session or cookies.

**public function viaRemember();**

**---**Log user in via Remember, Determine if the user was authenticated via ”remember me” cookie.

* used to checked whether the use logged in via the “remember me” cookie, while he attempts to change the password or attempt to view credit/debit card information

**public function logout()**

**Auth::logout()**

--Log user out, log user out of the application.

2019

Authentication User Experience

If the user is logged in we will show our dropdown menu and if they are logged out we will display a simple button to login.

We use blade @if statements and Auth::check() helper to see if the user is logged in. If they are logged in then our Auth::check() will return true and execute the if statement, otherwise it will execute the else statement.

when the user is logged in. We will have the logged in menu say "Hello Name". To do this, we take advantage of our second helper Auth::user() and then we just access the property for each column name in the user database table.

**Partials > \_nav.blade.php**

@if(**Auth::check()**)

<a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">

Hello {{ **Auth::user()->name** }}

</a>

<div class="dropdown-menu" aria-labelledby="navbarDropdown">

<a class="dropdown-item" href="{{ route('posts.index') }}">Posts</a>

<li>

<a href="{{ route('logout') }}"

onclick="event.preventDefault();

document.getElementById('logout-form').submit();">

Logout

</a>

<form id="logout-form" action="{{ route('logout') }}" method="POST" style="display: none;">

{{ csrf\_field() }}

</form>

</li>

</div>

@else

<a href="{{ route('login') }}" class="btn btn-default">Login </a>

@endif

# **Password Reset Emails**

We have migration for password reset at database>>migrations>>create\_passwrod\_resets\_table.php

Controllers>Auth>ResetPasswordController.php

We have

public function \_\_construct()

{

$this->middleware('guest');

}

Because we want only guest to send forgot password mail.

We don’t need logged in user resetting their password.

**Route (not needed in Laravel 5)-:**

// Password reset routes

Route::get(‘password/reset/{token?}’, ‘Auth\PasswordController@showResetForm’);

//? specifies token is optonal can present or not, both cases

Route::post(‘password/email’, ‘Auth\PasswordController@sendResetLinkEmail’);

Route::post(‘password/reset’, ‘Auth\PasswordController@reset’);

**Views (Not Needed in Laravel5):-**

Create views>auth>password>email.blade.php

Create views>auth>password>reset.blade.php

**We need to create a view for our email format** [auth.emails.password] **(Not Needed in Laravel5):-** *\*\*\*find the path in Laravel 5 for email format*

Create password.blade.php

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Click here to reset your password:<br>

<a href=”{{ $link = url(‘password/reset’, $token).’?email=’.urlencode($user->getEmailForPasswordReset()) }}”> {{ $link }}</a>

//everything after the ? mark is in the GET parameter. Laravel can automatically fill in the email address using the get parameter.

//If someone does not have html mail he can copy and paste the link provided into the url($link saved use from writing the url again )

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**\*We are using URL helper here**

**Creating the link for forgot my password in the login page.**

**//not required in laravel5**

views>auth>login.blade.php

<a href=”{{ url(‘password/reset’) }}”>Forgot My Password</a>

Setting up email sending in Laravel

We have to set few parameters inside the environment file .this is the same thing we did when we set up our database. This environment file is not sent to the git hub it is hidden for git hub. It contains all those parameters that make our application unique and we need to keep it secret.

Go to .env file in the base of our project.

Laravel using a powerful mail sending library known as **swift mailer** while sending mails.

SMTP setting for Gmail in Laravel .env files

MAIL\_DRIVER=smtp

MAIL\_HOST=smtp.gmail.com

MAIL\_PORT=25

MAIL\_USERNAME=pawargaurav49@gmail.com

MAIL\_PASSWORD=yourpassword

MAIL\_ENCRYPTION=tls

SMTP setting for mailtrap.io (A fake SMTP server to test mailing) in Laravel .env file:-

MAIL\_DRIVER=smtp

MAIL\_HOST=smtp.mailtrap.io

MAIL\_PORT=2525

MAIL\_USERNAME=0919fecbd1741a

MAIL\_PASSWORD=d75072b8ada951

MAIL\_ENCRYPTION=tls

Setting Default sending mailing address:-

config>mail.php

set ‘from’ field to send the mail from

Restart the server.

easiest way to set up authentication in a new Laravel application

we will be setting up a brand new Laravel application from scratch and a new database and then running our command to build the authentication

Creating new application name authentication for it:-

>Laravel new authentication

>cd authentication

Create a new database for the new application and put it credentials in the .env file of the Laravel application.

Create all the default authentication( sets up the controllers, migrations, and views for our application)

>php artisan make:auth

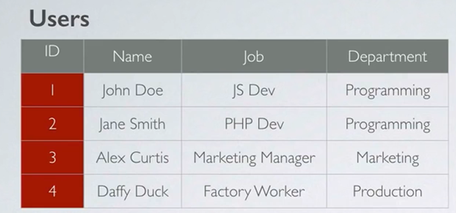
Runs migration to the database

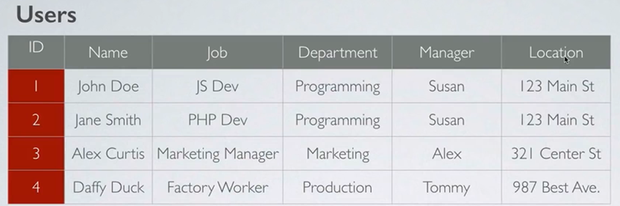
>php artisan migrate

Relationships in Laravel

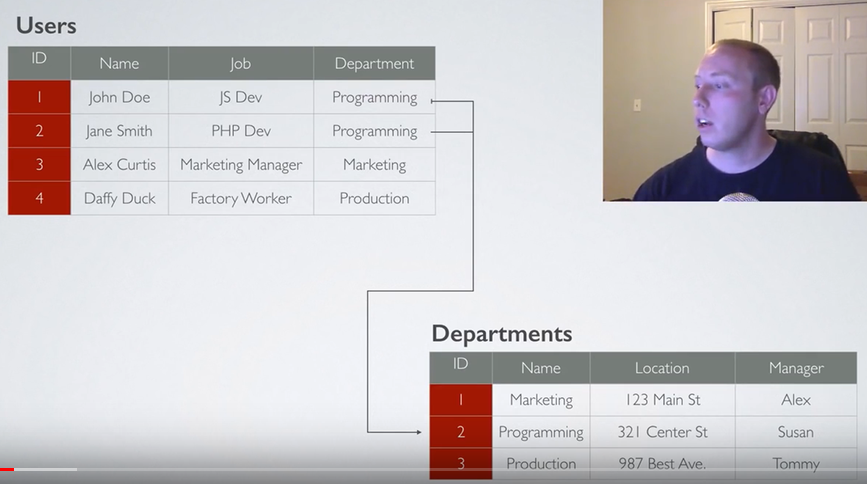
(Working with relational databases in Laravel to build Blog Categories)

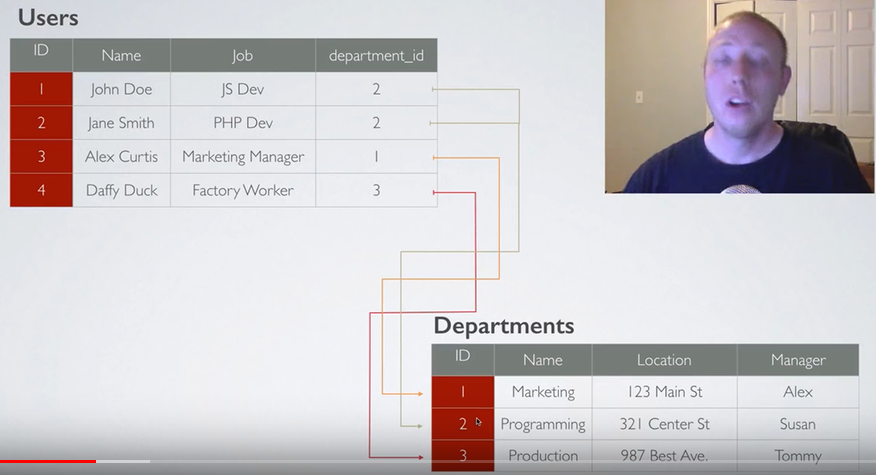
Relationship in database





Manager and Location are in relation with the Department, i.e., same department have same manager and same location, it causes repetition in the table if we have a large number of users, updating task would be repetitive e.g. in case of manager changes.





We call the Department as department\_id(name of the relational table\_id), id is unique.

We can change the name of department without effecting the relationship. (Dynamic Relationship)

Creating a table migration to store categories in database.

Category table to store category and category\_id column in posts table to link the category table to posts table.

Creating category model: -

php artisan make:model Category ---Singlular form and capital letter in begining

php artisan make:migration create\_categories\_table --create=categories

(we are using **--create=categories** it automatically creates table and adds a schema, otherwise we will be getting a blank migration page)

Adding the name of the category in migration file: -

public function up()

{

Schema::create('categories', function (Blueprint $table) {

$table->increments('id');

$table->string('name')

$table->timestamps();

});

}

Creating a column in the posts table that link to category table for each post(category\_id).

>php artisan make:migration add\_category\_id\_to\_posts --table=posts

(we use **--table=post**, because we are working on the existing table and not creating a new one)

(InnoDB table doesn’t support foreign key references in database level, so we are handling it on Laravel level, we will be hardcoding foreign key relation into the PHP)

//foreign key code does not do anything on the InnoDB table

public function up()

{

Schema::table('posts', function (Blueprint $table) {

$table->integer('category\_id')->nullable()->after('slug ')->unsigned();

//$table->foreign('category\_id')->references('id')->on('categories');

});

}

In case we want to rollback our column

public function down()

{

Schema::table('posts', function (Blueprint $table) {

$table->dropColumn('category\_id');

});

}

Run the migration

>php artisan migrate

//It will create categories table and update posts table and add category\_id column.

Manually telling Category model to use categories table: -

When we have difference in table name spellings, we need to do it manually otherwise it will take it by default.

……

class Category extends Model

{

protected $tables = 'categories';

}

…

//we are manually telling Laravel to use the categories table when we are working this model.

Defining the relationships :-

**One to many relationships :** We are defining that one category can have many posts. Inside Cateogory model…category.php

class Category extends Model

{

protected $tables = 'categories';

public function posts()

{

return $this->hasMany('App\Post');

}

}

//Category have one to many relationships with Post Model

//if we won’t follow the convention to name the foreign key we have to define the column name in the relationship. We follow the convention and named our column in posts table as category\_id

We now set the other half of relationship. We go to post model and define that posts are going below category. Inside Post model file posts.php:-

…

class Post extends Model

{

public function category()

{

return $this->belongsTo('App\Category');

}

}

Now create some categories into the database manually and manually connect give each post a category\_id.

Changing View to show Categories: -

single.blade.php –

….

<div class="col-md-8 col-md-offset-2">

…

<hr>

<p>Posted In: {{ $post->category->name }}</p>

</div>

//Using relationship, it will pull the name of the category form the database by Post model object $post.

//We can pull any value of categories table using this way.

Categories CRUD

Creating CRUD operation to create, Read, Update and Delete Categories.

Creating controller for our categories: -

>php artisan make:controller CategoryController --resource

//Camel case, Name\_of\_ModelController

//--resource will make it as CRUD controller.

We want to lock down categories so that only logged in users can create categories: --

For this we use the middleware in construct inside our CategoryController.

public function \_construct(){

$this->middleware('auth');

}

**use App\Category;** *//To use Category model in CategoryController*

In the index function we are pulling all the categories form the DB and passing it to the **categories.index** view.

public function index()

{

//Display view of all of our categories

//It will also have a form to create a new category, therefore we do not need the create function.

$categories = Category::all();

return view('categories.index')->withCategories($categories);

}

Setting up **routes** for categories CRUD:

Route::resource('categories', 'CategoryController');

//This give us all the additional routes for all the CRUD operations.

We don’t want categories.create route because we are creating our categories in index page only via a small form there. Therefore we can also delete our category() in CategoryController.

To stop showing the category.create route in >php artisan route:list

We are going to tell the Laravel to not to create a route for the create action inside the categories controller.we add a third parameter(array) to the route.

### Route::resource('categories', 'CategoryController', ['except' => ['create']]);

### //This give us all the additiona routes for all the crud functions except create.

Or we can use **‘only’** as third parameter in place of **‘except’** and it will create route for only those functions we want.

**Creating Navigation for categories:**

views>partials>\_nav.blade.php

<li><a href="{{ route('categories.index') }}">Categories</a></li>

**Creating view for categories:-**

Create a folder inside view for categories. -

Create view>categories>index.blade.php

**Create a from for adding new category in index.blade.php and call categories.store action in it.**

**Store the new categories into the database by using the store function of CategoresController :-**

use Session; // to use session functionality.

public function store(Request $request)

{

//Save a new category and redirect back to index

$this->validate($request, array(

'name' => 'require||max:255'

));

$category = new Category;

$category->name = $request->name;

$category->save();

Session::flash('success', 'New Category has been Created');

return redirect()->route('categories.index');

}

**Adding Category to Blog Post**:-

We are going to work with PostController for this.

Adding ability to add category while creating post:-

We are grabbing all the categories in the create function and showing them while creating new posts as dropdown.

App>Http>Controllers>PostController

use App\Category;

…

public function create()

{

$categories = Category::all();

return view('posts.create')->with('categories', $categories);

//or

//return view('posts.create')->withCategories($categories);

}

**Rendering Categories in the views:-**

resources>views>posts>create.blade.php

{{ Form::label('category\_id', 'Category:')}}

<select class="form-control" name="category\_id">

@foreach($categories as $category)

<option value="{{ $category->id }}">{{ $category->name }}</option> @endforeach

</select>

**Storing the form data with category\_id in store method of PostController**

public function store(Request $request)

{

//Validate the data

$this->validate($request, array(

'title' => 'required|max:255',

'slug' => 'required|alpha\_dash|min:5|max:255|unique:posts,slug',

**'category\_id' => 'required|integer',**

'body' => 'required'

)

);

//Store in the database

$post = new Post;

$post->title = $request->title;

$post->slug = $request->slug;

**$post->category\_id = $request->category\_id;**

$post->body = $request->body;

$post->save();

**resources>views>posts>show.blade.php**

<dl class="dl-horizontal">

<label>Category:</label>

<p>{{ $post->Category->name }}</p>

</dl>

**Creating ability to edit Categories:-**

Passing all the categories from the edit method of PostController to edit view :

public function edit($id)

{

//find the post in the database and save it as a variable.

$post = Post::find($id);

**$categories = Category::all();**

**$cats = array();**

**foreach ($categories as $cats) {**

**$cats[$category->id] = $category->name;**

**}**

//return the view and passin that variable

return view('posts.edit')->with('post', $post**)->with('categories', $cats);**

}

**posts>edit.blade.php**

**We are usng model-form binding in this form already:**

**{{ Form::label('category\_id', 'Category:', ['class'=>'form-spacing-top']) }}**

**{{ Form::select('category\_id', $categories, null, ['class' => 'form-control']) }}**

\*creating a select form in form-helper

{{ Form::select(‘category\_id’, array(‘key’ => ‘value’, ‘key2’ => ‘value2’, ‘key3’ => ‘value3’), default, array(other classes)) }}

\*key = value of select form

value = User display

<select value=”**key**”>**Userdispley**</select>

\*default is null because(is should be equal to id of current category) it should automatically take default category\_id because of model-from binding.

**Updating the category\_id in the update function of PostController:-**

public function update(Request $request, $id)

{

//Validate the data

$post = Post::find($id);

if($request->input('slug') == $post->slug){

$this->validate($request, array(

'title' => 'required|max:255',

**'category\_id' => 'required|integer',**

'body' => 'required'

));

} else{

$this->validate($request, array(

'title' => 'required|max:255',

'slug' => 'required|alpha\_dash|min:5|max:255|unique:posts,slug',

**'category\_id' => 'required|integer',**

'body' => 'required'

));

}

//Save the data to the database

$post = Post::find($id);

//or $post->title = $request->title;

$post->title = $request->input('title');

$post->slug = $request->input('slug');

**$post->category\_id = $request->input('category\_id');**

//of $post->body = $request->body;

$post->body = $request->input('body');

$post->save();

//set flash data with success message

Session::flash('success', 'The blog post was updated successfully saved!');

//Redirect whth flash data to posts.show

return redirect()->route('posts.show', $post->id);

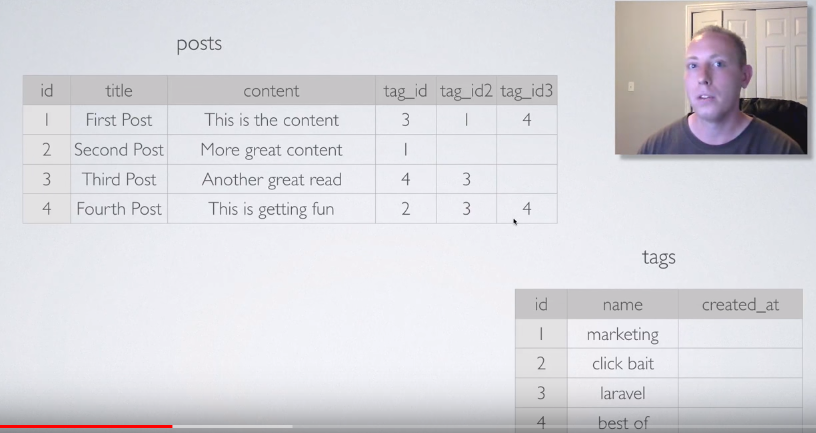
}

MANY-TO-MANY RELATIONSHIPS:-

**Problem: -**

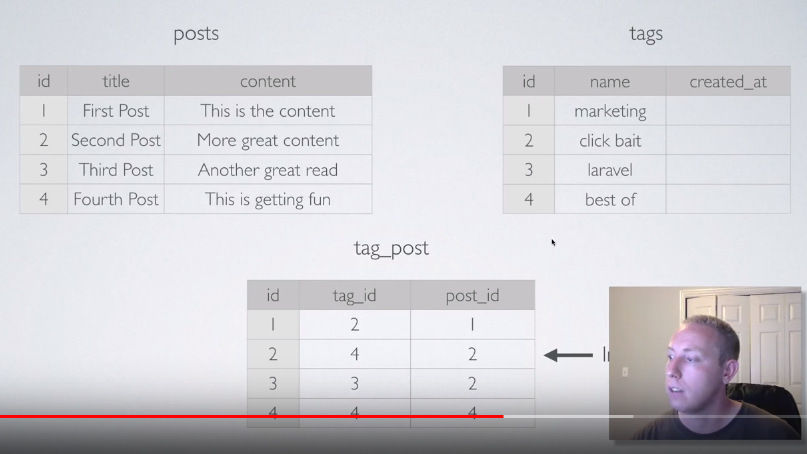
We can have many tags for a particular post. And a tag can be associated with multiple posts.

Ineffective method to address the issue: -



**Solution:-**

We need an Intermediary table: -



How do we create a many-to-many relationship in Laravel?

**belongsToMany();**

**SYNTAX:**

belongsToMany(‘App\Post’, ‘post\_tag’, ‘tag\_id’, ‘post\_id’);

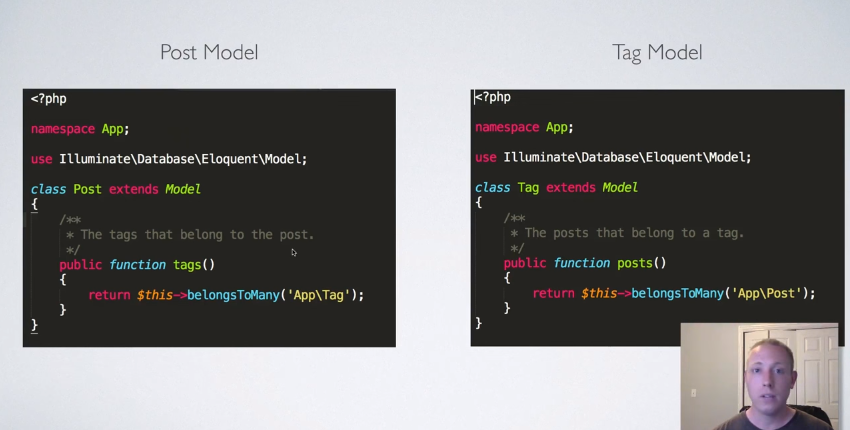
Name of Intermediary Table (both

table name in alphabet order joined by ‘\_’ in Laravel) (Suggested by Laravel, no need if we following the structure in naming intermediary table.)

Model to Link (Required)

Column name of Current Model(optional)

Column name for joining model(optional)



Building Tag Model:-

**Associating our post model with Tag model:**

**post.php**

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

…

**public function tags()**

**{**

**return $this->belongsToMany('App\Tag');**

**}**

}

**Creating Tag model: -**

>php artisan make:model Tag --migraton //capital First letter and single version of name of table

// --migration option will generate migration with model

*Model created successfully.*

*Created Migration: 2019\_03\_11\_014423\_create\_tags\_table*

**Creating many-to-many relationship with Post model:-**

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model

{

**public function posts()**

**{**

**return $this->belongsToMany('App\Post');**

//we are following the convention so we dont need to specify extra peramaetrs like intermediary\_table\_name, column\_id.

**}**

}

**Creating migration for tags Table :**

*2019\_03\_11\_014423\_create\_tags\_table.php –*

public function up()

{

Schema::create('tags', function (Blueprint $table) {

$table->increments('id');

**$table->string('name');**

$table->timestamps();

});

}

>**php artisan migrate**

*Migrating: 2019\_03\_11\_014423\_create\_tags\_table*

*Migrated: 2019\_03\_11\_014423\_create\_tags\_table*

**Creating migration for intermediary table “post\_tag”: -**

>php artisan make:migration create\_post\_tag\_table

*Created Migration: 2019\_03\_11\_020945\_create\_post\_tag\_table*

*2019\_03\_11\_020945\_create\_post\_tag\_table.php*

*public function up()*

*{*

*Schema::create('post\_tag', function (Blueprint $table) {*

*$table->increments('id');*

***$table->integer('post\_id')->unsigned();***

***$table->foreign('post\_id')->references('id')->on('posts'); //foreign key***

***$table->integer('tag\_id')->unsigned();***

***$table->foreign('tag\_id')->references('id')->on('tags'); //foreign key***

*$table->timestamps();*

*});*

*}*

*>php artisan migrate*

*Migrating: 2019\_03\_11\_020945\_create\_post\_tag\_table*

*Migrated: 2019\_03\_11\_020945\_create\_post\_tag\_table*

Tag CRUD

C:\Users\hp\Desktop\Laravel\blog>php artisan make:controller TagController --resource

Controller created successfully.

Creating Middleware so that only authenticated user have access to TagController: -

….

use App\Tag;

use Session;

….

public function \_construct()

{

$this->middleware('auth');

}

….

public function index()

{

$tags = Tag::all();

return view('tags.index')->with('tags', $tags);

}

…..

public function store(Request $request)

{

$this->validate($request , array('name' => 'required|max:191'));

$tag = new Tag;

$tag->name = $request->name;

$tag->save();

Session::flash('success', 'New Tag created Successfulluy!');

return redirect()->route('tags.index');

}

Route for tags: -

//Tags

Route::resource('tags', 'TagController', ['except' => ['create']]);

View for tags: -

resournce>views>tags>index.blade.php

Link for tags: -

partials > \_nav.blade.php

<li><a href="{{ route('tags.index') }}">Tags</a></li>

# **Adding Tag UI/UX**

Multi-Select HTML elements are traditionally very un-intuitive for users. In this part of our Laravel Blog Series we will create a better multi-select tool using a plugin called Select2 that runs off of Javascript.

<https://select2.org/>

We took the **select2.min.css** and **select2.min.js** from the folder we download (dist directory) and copied the files inside the CSS and js folder of public folder of Laravel installation.

We are adding the select2.min.css only to the page where it is needed

**posts > create.blade.php**

…

@section('stylesheets')

{{!! Html::style('css/parsley.css') !!}}

**{{!! Html::style('css/select2.min.css') !!}}**

@endsection

…

**{{ Form::label('tags', 'Tags:')}}**

**<select class="form-control select2-multi" name="tags[]" multiple="multiple">**

**@foreach($tags as $tag)**

**<option value="{{ $tag->id }}">{{ $tag->name }}</option>**

**@endforeach**

**</select>**

…..

…..

@section('script')

{{!! Html::script('js/parsley.min.js') !!}}

**{{!! Html::script('js/select2.min.js') !!}}**

@endsection

**<script type="text/javascript">**

**$('.select2-multi').select2();**

**</script>**

\*select2() is not workgin on my project

**posts>edit.blade.php**

…

@section('stylesheets')

{{!! Html::style('css/select2.min.css') !!}}

@endsection

……

@section('script')

{{!! Html::script('js/select2.min.js') !!}}

@endsection

**PostController.php :-**

Fetching the value of Tags form PostController

…

use App\Tag;

…

public function create()

{

$categories = Category::all();

**$tags = Tag::all();**

return view('posts.create')->with('categories', $categories)**->with('tags', $tags)**;

//or

//return view('posts.create')->withCategories($categories)->withTags($tags);

}

**Storing the tags values while creating a new post: -**

**store() of PostController**

dd($request); = dump and dye, displays all the values coming from the $request parameter and displays them and stops the program execution after it. It’s used to check the values we are getting.

Attaching the association for synching the tags to the corresponding post in post\_tag table:

**$post->tags()->sync($request->tags, false);**

sync() = It is a Laravel helper , it crates the relationship and syncs that up, parameter1 – array of all the item that we want to attach, Parameter2 – false .. default is true which will override the existing association, false tells Laravel to don’t override existing associations.

**Creating interface to view all tags: -**

posts>show.blade.php

**…**

<hr>

<div class="tags">

@foreach($post->tags as $tag)

<span class="badge badge-info">{{$tag->name}}</span>

@endforeach

</div>

**Creating interface to edit tags in a post :**

**PostController > edit()**

public function edit($id)

{

//find the post in the database and save it as a variable.

$post = Post::find($id);

$categories = Category::all();

$cats = array();

foreach ($categories as $category) {

$cats[$category->id] = $category->name;

}

**$tags = Tag::all();**

**// dd($tags);**

**$tags2 = array();**

**foreach ($tags as $tag) {**

**$tags2[$tag->id] = $tag->name;**

**}**

//return the view and passing that variable

return view('posts.edit')->with('post', $post)->with('categories', $cats)**->withTags($tags2)**;

}

**posts>edit.blade.php**

@section('stylesheets')

{{!! Html::style('css/select2.min.css') !!}}

@endsection

…

{{ Form::label('tags', 'Tags:', ['class' => 'form-spacing-top']) }}

{{ Form::select('tags[]', $tags, null, ['class' => 'select2-multi', 'multiple' => 'multiple'] ) }}

…

@section('script')

{{!! Html::script('js/select2.min.js') !!}}

<script type="text/javascript">

$('.select2-multi').select2();

$('.select2-multi').select2().val({{!! json\_encode($post->tags()->allRelatedIds()) !!}}.trigger('change'));

</script>

@endsection

Used for jquery plugin -> select(), so that the previously saved tags comes as selected on the form.

This plugin is initiated after the page already loaded in the font-end, we are not getting tags inputted automatically, so we need to manually do that.

Laravel use php to pass default values for all other elements but it won’t do it form select() jquery plugin.

allRelatedIds() = It’s a Laravel helper that gets all the Id’s number from the tags.

Updating all changes to the Database: -

PostController.php > update()

…

//Synching with post\_tab table

$post->tags()->sync($request->tags, true);

….

**$request->tags** is a array of the tag\_id’s which we are passing from the update form.

**true** will delate all the old relationship associated with that post\_id which we are updating and create new relationship between post\_id and updated tag\_id in the post\_tag table.

**show, edit and Update Tag CRUD: -**

TagController > show()

public function show($id)

{

$tag = Tag::find($id);

return view('tags.show')->with('tag',$tag);

}

views>tags>show.blade.php

Linking the Tag names to show the particular tag details: -

<td><a href="{{ route('tags.show',$tag->id) }}">{{ $tag->name }}</a></td>

views>tags>edit.blade.php

{{ Form::model($tag, ['route' => ['tags.update', $tag->id], 'method' => "PUT"]) }}

{{ Form::label('name', "Name:") }}

{{ Form::text('name', null, ['class' => 'form-control']) }}

{{ Form::submit('Save Changes', ['class' => 'btn btn-success', 'style' => 'margin-top:20px']) }}

{{ Form::close() }}

we are using Form::model() instead of Form::open(), because we want model-from binding through it. It takes the Tag model we have and going to pass it into the form and fill it in.

PostController edit() function : -

public function edit($id)

{

$tag = Tag::find($id);

return view('tags.edit')->with('tag',$tag);

}

PostController update() function : -

public function update(Request $request, $id)

{

$tag = Tag::find($id);

$this->validate($request, ['name' => 'required|max:255']);

$tag->name = $request->name;

$tag->save();

Session::flash('success', 'Tag name Successfulluy Updated!');

return redirect()->route('tags.show', $tag->id);

}

# **Deleting Tags Safely**

Deleting many-to-many relationships in Laravel 5.2 means that we must use a special detach() method before deleting the resource itself.

In previous videos we worked on our Tags model and the CRUD involved with it. We set up our pivot table to track the relationship between tags and posts. Tags have many posts and posts have many tags.

But deleting these relationships can be difficult. We can't simply delete the tag because then the posts will still be looking for the tag, but will get an error because the tag does not exist.

If a post is removed, then the tags will point to posts that do not exist. So we must modify the delete methods for both posts and tags to make sure that these links between tags and posts are deleted when a resource is deleted to reflect the new structure.

We will use the detach() method to remove these links before deleting the resource. This method makes our jobs easy because it does all the hard work. It removes all the links for the resource we are deleting. If we run it on a post object then it will remove all the links to posts equal to the one we are deleting from the table. If we run it on a tag object then it will remove all links for that tag from the post\_tag table.

Controllers > PostController

public function destroy($id)

{

$post = Post::find($id);

**$post->tags()->detach();**

$post->delete();

Session::flash('success','The post was successfully deleted.');

return redirect()->route('posts.index');

}

Controller > TagsController > destroy()

public function destroy($id)

{

$tag = Tag::find($id);

//detach() is used to detach the relationship that this tag is having with posts in post\_tag table

$tag->posts()->detach();

$tag->delete();

Session::flash('success', 'Tag has been deleted Successfulluy');

return redirect()->route('tags.index');

}

views>tags>show.blade.php

…..

<div class="col-md-2 ">

{{ Form::open(['route' => ['tags.destroy', $tag->id], 'method' => 'DELETE']) }}

{{ Form::submit('Delete', ['class' => 'btn btn-danger btn-block', 'style' => 'margin-top:10px;']) }}

{{ Form::close() }}

</div>

…..

# **Sending Email from Contact Form:**

Mailer class in Laravel is built on the php swift mailer library.

.env file will override the setting of config>mail.php file.

route to go to contact form >

Route::get('/contact','PagesController@getContact');

route to send email from contact form >

Route::post('/contact','PagesController@postContact');

PagesController.php >

…

use Illuminate\Http\Request;

use Mail;

use Session;

…

public function getContact(){

return view('pages.contact');

}

public function postContact(){

//validation

$this->validate($request, [

'subject' => 'min:3',

'email' => 'required|email',

'message' => 'min:10'

]);

$data =array(

'email' => $request->email,

'subject' => $request->subject,

'bodyMessage' => $request->message

);

//Laravel 5.2

Mail::send('emails.contact', $data, function($message) use ($data){

$message->from($data['email']);

$message->to('hello@gauav.com');

$message->subject($data['subject']);

});

Mail::to('hello@gaurav.com')->send(new ContactForm($request));

or

//Mail::to('hello@gaurav.com')->send(new ContactForm($data));

Session::flash('success', 'Your email was Sent!');

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

return redirect(‘/’);

}

Views>pages>contact.blade.php

…

<form action="{{ url('contact') }} method="POST">

…

we are using URL helper because we didn’t set the named routes form /contact, because we don’t need named route to use here like route(‘contact’)

**Syntax of Mail function**

*Mail::send('view', $data, function(){ });*

view – the view we want to send it to.

$data – the data we want to pass into the view

function() – closure function that contain all the header information for the email, to, from , bcc, cc etc.

Laravel 5.5 mail –

[Generating Mailables](https://laravel.com/docs/5.5/mail" \l "generating-mailables) –

>php artisan make:mail ContactForm

Mail created successfully.

app>Mail>ContactForm.php

….

public function \_\_construct($data)

{

$this->data = $data;

}

/\*\*

\* Build the message.

\*

\* @return $this

\*/

public function build()

{

return $this->from('bhaskar@example.com')

->view('emails.contact');

}

….

Controllers>PagesController.php

public function postContact(Request $request){

//validation

$this->validate($request, [

'email' => 'required',

'subject' => 'min:3',

'message' => 'min:10'

]);

Mail::to('hello@gaurav.com')->send(new ContactForm($request));

Session::flash('success', 'Your email was Sent!');

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

return redirect('/');

}

views>emails>contact.blade.php

<h3>You have a new contact via the Contact Form</h3>

<div>

Subject :

<h2> {{ $data->subject }} </h2>

Msage:

{{ $data->message }}

</div>

<p>Email : {{ $data->email }}</p>

# **Adding Comments : -**

Model and migration for comments –

>php artisan make:model Comment --migration

Model created successfully.

Created Migration: 2019\_03\_20\_233611\_create\_comments\_table

database > migrations > create\_migration\_table.php

…

class CreateCommentsTable extends Migration

{

public function up()

{

Schema::create('comments', function (Blueprint $table) {

$table->increments('id');

$table->string('name');

$table->string('email');

$table->text('comment');

$table->boolean('approved');

$table->integer('post\_id')->unsigned();

$table->timestamps();

});

Schema::table('comments', function($table){

$table->foreign('post\_id')->references('id')->on('posts')->onDelete('cascade');

});

}

/\*\*

\* Reverse the migrations.

\*

\* @return void

\*/

public function down()

{

Schema::dropForign(['post\_id']);

Schema::dropIfExists('comments');

}

}

>php artisan migrate

Migrating: 2019\_03\_20\_233611\_create\_comments\_table

Migrated: 2019\_03\_20\_233611\_create\_comments\_table

Models: -

app>Post.php

…

public function comments()

{

return $this->hasMany('App\Comment');

}

…

app>Comment.php

…

public function post()

{

return $this->belongsTo('App\Post');

}

…

CONTROLLER: -

>php artisan make:controller CommentsController –resource

Controller created successfully.

**Crating route for storing comment :**

Route::post('comments/{post\_id}', ['uses' => 'CommentsController@store', 'as' => 'comments.store']);

**Creating form to add comment :**

views > blog > single.blade.php

**Setting the store() of Comment Controller:**

Controllers> CommentsController.php

public function store(Request $request, $post\_id)

{

$this->validate($request, [

'name' => 'required|max:255',

'email' => 'required|email|max:255',

'comment' => 'required|min:5|max:2000'

]);

$post = Post::find($post\_id);

$comment = new Comment;

$comment->name = $request->name;

$comment->email = $request->email;

$comment->comment = $request->comment;

$comment->approved = true;

$comment->post()->associate($post);

$comment->save();

Session::flash('success', 'Comment was added Successfully');

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

return redirect()->route('blog.single', [$post->slug]);

}

**Editing Blog to view user comments :-**

**views>blog>single.blade.php**

**…**

<div class="row">

<div class="col-md-8 col-md-offset-2" >

<h3 class="comment-title"> <img src="/comment-square.svg" alt="comment-square"> {{ $post->comments->count() }} Comments</h3>

@foreach($post->comments as $comment)

<div class="comment">

<div class="author-info">

<img src="{{ **"https://www.gravatar.com/avatar/".md5( strtolower( trim( "$comment->email" ) ) )."?d=monsterid"** }}" class="auther-image" >

<div class="auther-name">

<h4>{{ $comment->name }}</h4>

<p>{{ date('d/M/Y - g:iA',strtotime($comment->created\_at)) }}</p>

</div>

</div>

<div class="comment-content">

{{$comment->comment}}

</div>

</div>

@endforeach

</div>

</div>

**…**

//using gravator to show the image of the person in comments who have set the global gravatar image and information in wordpress gravatar.

Adding the ability to manage the comments from the back end

Making view to edit and delete comment: -

views>posts>show.blade.php

…

<div id="backend-comments" style="margin-top: 50px;">

<h3>Comments<small>{{ $post->comments()->count() }} total</small></h3>

<table class="table">

<thead>

<tr>

<th>Name</th>

<th>Email</th>

<th>Comment</th>

<th width="150px">Action</th>

</tr>

</thead>

<tbody>

@foreach($post->comments as $comment)

<tr>

<td>{{ $comment->name }}</td>

<td>{{ $comment->email }}</td>

<td>{{ $comment->comment }}</td>

<td>

<a href="{{ route('comments.edit', $comment->id) }}" class="btn btn-xs btn-primary"><i class="fa fa-pencil" aria-hidden="true"></i></a>

<a href="{{ route('comments.delete', $comment->id) }}" class="btn btn-xs btn-danger"><span class="fa fa-trash"></span></a>

</td>

</tr>

@endforeach

</tbody>

</table>

</div>

…

**Creating route for comments: -**

**//comments**

Route::post('comments/{post\_id}', ['uses' => 'CommentsController@store', 'as' => 'comments.store']);

**Route::get('comments/{id}/edit',['uses' => 'CommentsController@edit', 'as' => 'comments.edit']);**

**Route::put('comments/{id}',['uses' => 'CommentsController@update', 'as' => 'comments.update']);**

**Route::delete('comments/{id}',['uses' => 'CommentsController@destroy', 'as' => 'comments.destroy']);**

**CommentController -> edit()**

public function edit($id)

{

$comment = Comment::find($id);

retrun view('comment.edit')->with('comment', $comment);

}

**Creating view for edit comment**

resources>views>comments>edit.blade.php

**CommentController -> update()**

public function update(Request $request, $id)

{

$comment = Comment::find($id);

$this->validate($request, ['comment' => 'required|min:5|max:2000']);

$comment->comment = $request->comment;

$comment->save();

Session::flash('success', 'Comment Updated Successfully!!');

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

return redirect()->route('posts.show', $comment->post->id);

}

Deleting the Comment:

**show.blade.php**

**…**

<a href="{{ route('comments.delete', $comment->id) }}" class="btn btn-xs btn-danger"><span class="fa fa-trash"></span></a>

**…**

**Createing delete route for the get request of button(\*this is different then real destroy method)**

Route::get('comments/{id}/delete',['uses' => 'CommentsController@delete', 'as' => 'comments.delete']');

**Creating Controller for delete method in CommentConroller :**

public function delete($id)

{

$comment = Comment::find($id);

return v**i**ew('comments.delete')->with('comment', $comment);

**}**

**Crating view for comment.delete in view>comments>delete.blade.php**

**…**

**{{ Form::open(['route' => ['comments.destroy', $comment->id], 'method' => 'DELETE']) }}**

**{{ Form::submit('Yes Delete This Comment', ['class' => 'btn btn-danger btn-block']) }}**

**{{ Form::close()}}**

**…**

**Creating CommentController for Destroying Comment:**

public function destroy($id)

{

$comment = Comment::find($id);

$post\_id = $comment->post->id;

$comment->delete();

Session::('success', 'Comment Deleted Successfully!!!');

return redirect()->route('posts.show',$post\_id);

}

**Setting up authentication middleware in CommentController so that only authenticated login user can delete the comments -**

public function \_\_construct()

{

$this->middleware('auth', ['except' => 'store']);

}

\*\*//to allow front end user to store comment without being authenticated we use second parameter ['except' => 'store']

# Adding a WYSIWYG Editor(What You See Is What You Get):-

Some of the editors available are

CK editor

**Tiny MCE**

WYSIWYG HTML5

Mercury

resources>views>posts>create.blade.php :-

Adding Tiny MCE Plugin using CDN:

<script src="https://cloud.tinymce.com/5/tinymce.min.js?apiKey=z1o0q9ftqjxl3bsdtjckevmt3m91mze79smmkr0haeqv9erj">

</script>

<script>tinymce.init({

selector:'textarea',

plugins: "link code",

});

</script>

Adding Tiny MCE editor to edit page:

Resources>views>posts>edit.blade.php

\*\*add as above

Dealing with the html code output in show.blade.php

resources>views>posts>show.blade.php

{{ $post->body }} -----------> {!! $post->body !!}

{{ }} = use HTML special cars function of php, it converts HTML tags to unexucatable characters

{!! !!} = execute the html code inside and displays output

Doing curly bang bang {!! for all the page

blog>single.blade.php

**Using strip\_tags() to strip all HTML tags in *pages :-***

*pages>welcome.blade.php*

*<p>{{ substr(****strip\_tags($post->body)****, 0, 80) }}{{ strlen(****strip\_tags($post->body)****) > 80 ? "...": ""}}</p>*

blog>index.blade.php

posts>index.blade.php

# We will be using a library called HTML Purifier

which works to clean up HTML code from our WYSIWYG editor to only allow certain HTML tags that we whitelist, and sanitize all the other tags so they can not execute. This means that if someone tries to put a script tag into our database, we will render it safe before it gets there, protecting our application.

**htmLawed is another one,**

**Use mewebstudio/Purifier, laravel plugin for html purifier**

**https://github.com/mewebstudio/Purifier**

This package can be installed via [Composer](http://getcomposer.org) by requiring the mews/purifier package in your project's composer.json:

{

"require": {

"laravel/framework": "~5.0",

"mews/purifier": "~2.0",

}

}

or

Require this package with composer:

composer require mews/purifier

Update your packages with composer update or install with composer install.

**>composer require mews/purifier**

**In PostController.php**

use Purifier;

**Store function:**

$post->body = **Purifier::clean(**$request->body**)**;

Update function:

$post->body = Purifier::clean($request->input('body'));

These are going to clean our HTML Code before saving it into the database.

\* $request->body is same as $request->input(‘body’);

HTML Purifier is removing <heading> tags and adding <p> tag due to the setting.

To use your own settings, publish config.

$ php artisan vendor:publish --provider="Mews\Purifier\PurifierServiceProvider"

edit file config/purifier.php

Adding h1,h2,h3,h4,h5,h6 to ‘HTML.Allowed’ and 'AutoFormat.AutoParagraph' => false,

Image Intervention, a Laravel package, to handle resizing and manipulating images submitted by users.

>composer require intervention/image

## Integration in Laravel

Setting **Service Provider and Facades-**

open your Laravel config file config/app.php and add the following lines.

In the $providers array add the service providers for this package.

Intervention\Image\ImageServiceProvider::class

Add the facade of this package to the $aliases array.

'Image' => Intervention\Image\Facades\Image::class

Modifying the view to upload the image:-

Views>posts>create.blade.php

{!! Form::open(['route' => 'posts.store', 'data-parsley-validate' => '', **'files' => true**]) !!}

…

**{{ Form::label('featured\_image', 'Upload Featured Image:') }}**

**{{ Form::file('featured\_image') }}**

**Creating migration to save image name to database**

>php artisan make:migration add\_image\_col\_to\_posts --table=posts

migrations>2019\_05\_03\_054310\_add\_image\_col\_to\_posts

public function up()

{

Schema::table('posts', function (Blueprint $table) {

**$table->string('image')->nullable()->after('slug');**

});

}

public function down()

{

Schema::table('posts', function (Blueprint $table) {

**$table->dropColumn('image');**

});

}

**>php artisan migrate**

**PostController.php**

use Image;

Store()

…

if($request->hasFile('featured\_image')){

$image = $request->file('featured\_image');

$filename = time().'.'.$image->getClientOriginalExtension();

$location = storage\_path('app/public/'.$filename);

Image::make($image)->resize(800,400)->save($location);

$post->image = $filename;

}

****Validating image:****

**'featured\_image' => 'sometimes|image'**

**Showing the image in blog**

Blog>single.blade.php

<img src="{{ asset('images/'.$post->image) }}"/>

**Or**

create a symbolic link from public/storage to storage/app/public

To create the symbolic link, you may use the storage:link Artisan command:

php artisan storage:link

<img src="{{ asset('storage/'.$post->image) }}" alt="" />

**Editing and Updating images:**

**Resources>views>posts>edit.blade.php**

{!! Form::model($post, ['route'=>['posts.update', $post->id], 'method'=>'PUT', **'files' => true**]) !!}

…

**{{ Form::label('featured\_image', 'Update Featured Image:', ['class' => 'form-spacing-top']) }}**

**{{ Form::file('featured\_image') }}**

**PostController.php**

use Image;

use Storage;

update()

**\*Inserting the $id at the unique force laravel not to check the slug against its own slug, when slug has been updated , while checking for unique slug.**

**$this->validate($request, array(**

**'title' => 'required|max:255',**

**'slug' => "required|alpha\_dash|min:5|max:255|unique:posts,slug,**$id**",**

**'category\_id' => 'required|integer',**

**'body' => 'required',**

****'featured\_image' => 'image'****

**));**

**…**

if($request->hasFile('featured\_image')){

//add the new photo

$image = $request->file('featured\_image');

$filename = time().'.'.$image->getClientOriginalExtension();

$location = storage\_path('app/public/'.$filename);

Image::make($image)->resize(600,400)->save($location);

$oldFileName = $post->image;

//update the database

$post->image = $filename;

//delete the old photo

Storage::delete($oldFileName);

}

We are using Storage Facade to delete the image, and Image facade to save the new image to the desired location.

\*In config>filesystem.php we are changing default File system disk form

**'default' => env('FILESYSTEM\_DRIVER', 'local'),**

to

**'default' => env('FILESYSTEM\_DRIVER', 'public'),**

Because we are storing our image file into the app/public folder of storage directory and public file-system disk root path is

'root' => storage\_path('app/public'),

**PostController.php**

destroy()

public function destroy($id)

{

$post = Post::find($id);

$post->tags()->detach();

**Storage::delete($post->image);**

$post->delete();

Session::flash('success','The post was successfully deleted.');

return redirect()->route('posts.index');

}

It will delete the image file before deleting the post correlating to the that image

We can show the photo in posts>show.blade.php

<img src="{{ asset('storage/'.$post->image) }}" alt="" />