

Laravel Fundamentals – Routes – Serving our apps

One thing to note before diving into this portion of the lecture make sure to pay attention, because later on you are going to see me use an extension for our projects, every time we create a project we create an extension. You are going to create a project with its name ending in .dev, This is a feature of Laravel.

As you remember, there are multiple ways of serving our content to the browser exist. One way I demonstrated this is with php artisan serve, which uses the built-in server provided by php. This means that your application will be available in <http://localhost:8000>. You can change where you server your application by using php artisan serve --port = 'enter port number'.

The other way is using 'homestead', homestead is a more complicated way of using Laravel. It is a virtual environment using Vagrant and VirtualBox 6.x. This can be a problem if your machine does not have virtualization enabled ex you need to have Hyper-V installed and working otherwise Its not going to work.

Valet is very similar to 'homestead' but is only available to mac users, this has already been covered in the mac portion of the installation, so refer to that for more information. But for now, serving our application locally will suffice.

A couple of differences are instead of running the app at the local level we run the application with our project name and an extension, and that we have to cycle the server every once in a while to see our matching changes in the viewport.

Laravel Fundamentals – Routes – Laravel Structure Overview

This section will introduce you to the Laravel structure and files. This information is platform agnostic, meaning that it will work for either mac, windows, and Linux.

- App - The main folder for the application, we store our models and our controllers
- Bootstrap - Stores the code that 'bootstraps' our application, meaning our entry point. Files and dependences are located and compiled with this file
- Config - Stores all of our configuration files for our application, for example, suppose you want to use sqlite instead of MySQL, you can change this in the configuration file. Another example is in mail.php you can specify what e-mail service you want to have, instead of SMTP you can switch to Mailgun. There is a lot of data in this folder so don't get overwhelmed most of these files will be gone over throughout this course
- Database - Stores our migrations and is where we create our tables with Laravel. It helps us create tables with Laravel, we create our database and then manage our tables within Laravel. We can also create heap files to store a large amount of data with the subdirectory factories and seeders

- **Public** - This is where our public-facing files will be located, examples include css, javascript, our index.php, and our htaccess file for security. It's also the main area of the application
- **Resources** – The files that need to be compiled using the vite.config.js (previously called webpack!?! I wonder why they changed this lol) allow us to compile multiple css, JavaScript and sass files into one css, JavaScript, and sass file. Lastly, we have our views, This is the html portion of our application, for example, if we have a contact page, this is where the data will be for that contact page, or an about page unless it pulls from the database, then we make that request to the controllers, then the controls make a request to the database. Whenever you think about controls think of the word middleman, this will be expanded upon in a later lecture. Once files are compiled in the Resources folder, it moves to the public folder.
- **Routes** - In the file web.php is where our routes are located, essentially this directs the user to a new page after a specific element or action takes place. In api.php is where we have endpoints that our not public, it is going to request some data from either the application from the user, or from an external database hosted somewhere on the web
- **Storage** - Ignore this for now
- **Tests** - Ignore this for now
- **Vendor** - This is a folder containing dependencies provided by the composer. If you explore this, you can see that Laravel is one of the frameworks installed by the composer

Now let's move on to the root files contained in our application.

- **.env** - This is where we contain all sensitive information, for example, suppose we want to connect to a database, we place our configuration information, including the address, port, password, and username. Another example says we want to change the sender address for our e-mail service, we can do that here. If you are connecting to aws to host your application, that information is stored here. Very sensitive information, when you push your project to a repository, as you can see in the file .gitignore this file is ignored and won't be public.
- **Artisan** - This file won't be edited, but it contains a program that we use to generate controllers, tables, controller classes, and generally most of the functionality of our application.
- **Composer.json** - Every time composer adds another package we have a reference to the version and additional dependencies that our packages use.
- **Package.json** - This is the same as composer.json except that it's for our npm packages, for example, react, nodejs, etc.

- Vite.config.js - This is a wrapper, formally called webpack it uses Laravel-vite-plugin to wrap around Laravel, basically this allows us to compile the app.css and app.js files into a singular css and js files located in resources.

I understand that this is a lot of information, but once we start working with this it will get easier and easier.

Laravel Fundamentals – Routes Part 1

In Laravel routes will either make or break your application from a user experience standpoint. Routes in their most basic form are the url on the address bar.

In this portion of the lecture, we will now explore in greater detail a Laravel project's routes. This portion of the lecture (unless otherwise stated is platform agnostic for mac and Windows).

First, open your example-app project and run - php artisan serve to make sure your application is functional.

Next navigate to the routes folder and open the web.php file. You should see the following code:

So the first thing that you are going to notice here is that we have this class here called Route: As you can see it contains a static method (you can tell it uses a static method because of the two colons) then we have a couple of parameters, first is the '/' parameter that lets users know what page we are currently on. This is done through string catenation, basically adding the route name to the end of the forward slash. So right now when we go to the root directory of this application, it will give us a view. Now remember what I mentioned about views, It is a representation of an HTML file compiled from our application. Everything that you see on the browser would be a view.

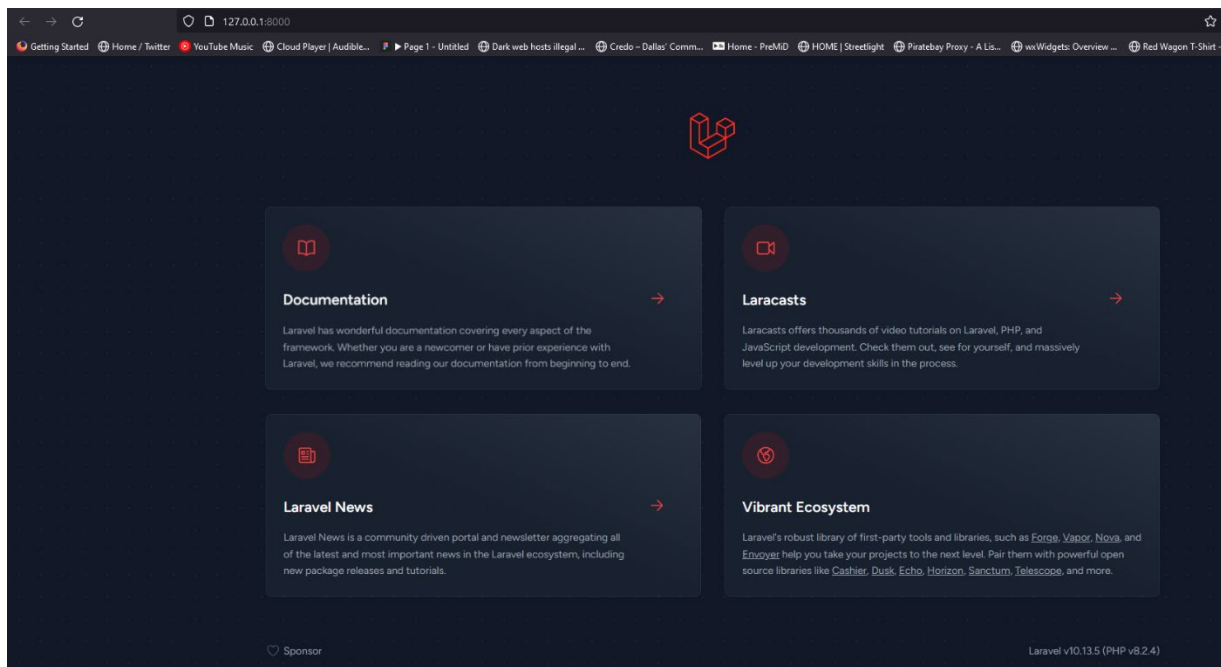
```
<?php

use Illuminate\Support\Facades\Route;

/*
|-----
|
| Web Routes
|-----
|
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "web" middleware group. Make something great!
*/
```

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

Now let us go to the browser for an example, the standard welcome view displays as follows:

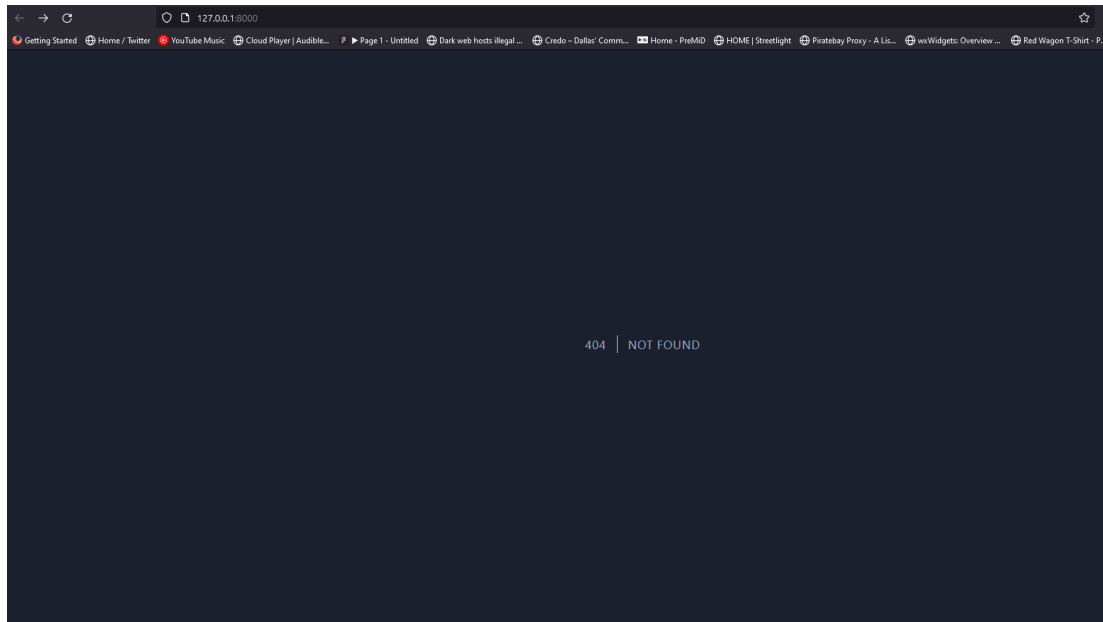


Now if we change our first parameter to:

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

And we change our address to <http://127.0.0.1:8000/example> we will see the same home screen as the root example. But this also means that if we try to go to the root address we will receive a 404 error as

shown below:



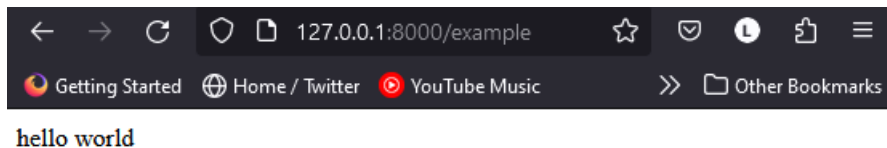
Next, add the following code below our get route to the web.php file. This is called a route group, it basically stores all of our routes in a container. This is done for security reasons and is also standard for the MVC model mentioned in the last lecture. In the Route signature, we have the string 'middleware'. Every time you see middleware it is because you are securing something there. So we are saying now that everything in this route group is going to be available to the web.php file. :

```
Route::group(['middleware' => ['web']], function () {  
});
```

All right to recap, our first parameter is where you want to go. Second parameter could be many different things, and we could even have a third parameter, but we will approach this later. Right now our Route::get class contains a closure function that is responsible for returning the welcome view to the user's viewport. Suppose you change the closure function as follows:

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

When you run the application, the following is displayed:



This is really cool!!! For context, with vanilla php to output a string to the viewport, we would first need to create a folder named index.php, put some php files in there, and run the `eco()` command, it would look something like this:

```
return eco(index.php/file.php);
```

Laravel allows us the freedom to forgo creating separate php route files to be displayed by a calling function, its essentially plug-and-play routes. If we wanted to we could even pass ID's and extra parameters to pass into the view and return it. Quite Incredible!

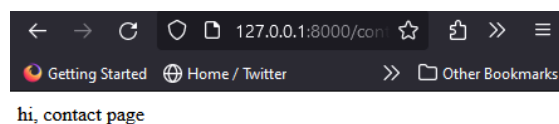
This is only the basics of what you can be with php routes, for now, let's fix our `Route::get` back to the following:

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

Laravel Fundamentals – Routes Part 2

Now that we have a basic understanding of routes. One might assume that It's super easy to create additional routes like so:

```
Route::get('/about', function () {  
    return "hi about page";  
});  
  
Route::get('/contact', function () {  
    return "hi, contact page";  
});
```

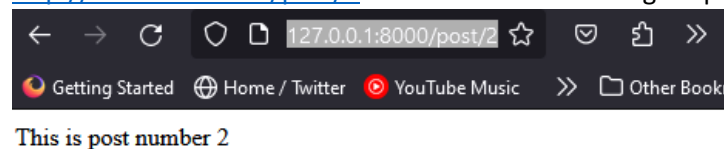


This outputs decently enough as we expected :

Now we will demonstrate how to pass the parameters to the views and do whatever you want inside our closure function, all while using one route closure function per route type. let's say for example you create another route as follows, we have our route with a static get() method, and then the first parameter with single quotation marks and a slash to indicate we want to go from route to whatever comes next. So now let's say for example we have a post, but it is post1 of many posts we have implemented, we would make our implementation as follows:

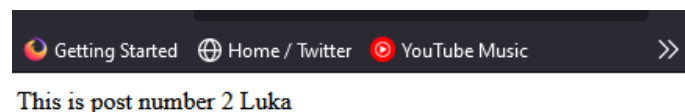
```
Route::get('/post/{id}', function($id){  
    return "This is post number ".$id;  
});
```

We concatenate the id at the end of the return string, and change the address to <http://127.0.0.1:8000/post/2> we receive the following output:



As you can see we are passing the address variable directly to the viewport. You can also pass multiple parameters like the following example:

```
Route::get('/post/{id}/{name}', function($id,$name){  
    return "This is post number ".$id." ".$name;  
});
```



Later on, I will demonstrate how we properly use this teaching of passing the variable: by passing it over to views and executing further logic from there.

Laravel Fundamentals – Naming Routes

Now so far we have studied how we can pass a parameter and return some information, and learned how to pass parameters to routes, now we will work on creating a standardized naming convention.

Let's try an example, suppose we have the following route that is relatively long address:

```
Route::get('admin/posts/example', function(){
```

```
});
```

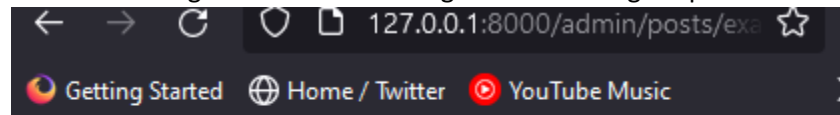
If you want to reduce the size of this address you can nest the closer function in an associative array and wrap the address into any string you desire like the following:

```
Route::get('admin/posts/example',array('as'=>'admin.home', function(){  
}));
```

We are not finished yet, to access this route we must use the following Laravel helper function:

```
Route::get('admin/posts/example',array('as'=>c, function(){  
    $url = route('admin.home');  
  
    return "this url is ".$url;  
}));
```

Now when we go the address at we get the following output:



this url is <http://127.0.0.1:8000/admin/posts/example>

So what this means is 'admin.home' = admin/posts/example, we can check this by closing the sever and typing the command into bash - php artisan route:list . We should receive the following output:

```
GET|HEAD / .....  
POST _ignition/execute-solution ignition.executeSolut..  
GET|HEAD admin/posts/example ..... admin.home  
GET|HEAD api/user .....  
GET|HEAD contact .....  
GET|HEAD post/{id}/{name} .....  
GET|HEAD sanctum/csrf-cookie sanctum.csrf-cookie > Larave..  
  
Showing [10] routes
```

Later in this lecture, we will demonstrate further advanced routing techniques but In summary, we are able to assign a variable to the URL address. This is a Laravel-specific example, in vanilla php this would be convoluted, requiring folders and separate files to achieve a similar effect. [Laravel Docs Routes \(https://laravel.com/docs/5.2/routing\)](https://laravel.com/docs/5.2/routing)

Laravel Fundamentals – Controllers – Introduction

In this section, we will learn and demonstrate Controllers in Laravel. Remember that controllers is a class or a group of classes that deal with information coming from the database and relaying or passing

information to a view or a representation of a page i.e. a contact page or reference page. For example let's say I represent a controller, I would receive information from the database, then pass it along to the view, this works in both directions meaning that information can be taken from the view and passed to the database accordingly

Laravel Fundamentals – Controllers

Throughout this section, we are going over controllers. So essentially controllers are the middleman. So where are they?

First, they are located in the controller's file inside the Http file. Now you can have subfolders for all the types of controllers, or you can just put them all in the controllers' subfolder. Laravel includes a main controller called Controller.php, go ahead and open it. This controller might be a little different from the rest but that is because this is the main controller. So let's explain a bit about some of the keywords that some of you might not be familiar with. (for reference keywords are reserved words that can not be used as variable names or parameter names, they serve a specific function unique to the programming language).

We have Namespace and use.

What is namespace: namespace does for functions and classes what scope does for variables. It basically allows us to use the same function names or class names in different parts of the same application without causing problems. So for example let's say I have 2 functions called 'runthisquery' in the same application. If I call the same function name in the same application we would have a problem.

With namespaces, what we can do is have a namespace like the one below:

```
namespace App\Http\functions1;
```

Now when we call the runthisquery function, we know the version from the functions1 namespace if we wanted to also the second version of this function, we could add another namespace like so:

```
namespace App\Http\functions2;
```

We use a namespace to differentiate the functions, even though they have name, so we don't have a collision (remove the functions2 namespace from the program).

Use is used to import that specific class or namespace to the scope of the file.

```
use Illuminate\Foundation\Validation\ValidatesRequests;
```

When I call this use function, I am able to use the class ValidatesRequests within the scope of the file.

So now let's move on to creating controllers. We can create a controller rather simply, first create a file named PostController.php in the controllers file directory and that's it, we have created a post controller.

Now in Laravel, there is an easier way to create a controller using the terminal(delete the controller we just made) in bash, type the following command - php artisan make:controller PostController

If the command has ran successfully you should receive the following output: `<?php`

```
namespace App\Http\Controllers;

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

class PostController extends Controller
{
    //
}
```

(delete the PostController) But the preferred method for creating controllers with resources, we will go into what resource means later on, but essentially it creates a controller with create, read, update, delete, etc. So to create a controller with all of the methods I listed, use the following bash command - `php artisan make:controller --resource PostController` .

You should have the following file (PostController.php)

```
class PostController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
```

```
{
    //

}

/**
 * Display the specified resource.
 */
public function show(string $id)
{
    //
}

/**
 * Show the form for editing the specified resource.
 */
public function edit(string $id)
{
    //
}

/**
 * Update the specified resource in storage.
 */
public function update(Request $request, string $id)
{
    //
}

/**
 * Remove the specified resource from storage.
 */
public function destroy(string $id)
{
    //
}
}
```

We can now control the information coming from the database and the information from the view, in the next section we will demonstrate how to use the controller.

In the last section, we demonstrated how to create a Laravel controller in different ways, first manually, then with the command line using php artisan. Now before we move on let me show you how to find some more commands in php artisan. In bash, if you type in the following - php artisan will display all of php artisan's commands.

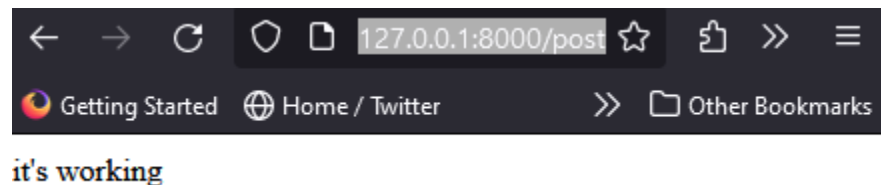
Now back to the post controller and use it as an example. Move back to the web.php file containing our route's, because as I mentioned earlier, this is where we start to build everything in our application. Start by commenting out all of the previous Route::get methods. And create another Route::Get request as seen below:

Instead of a callback function, we are going to use a controller instead, now we have to tell it what type of method this get is going to hit. To do this I want to access the index method. So once I do that then every time I click on post controller it is going to look for the index method.

Now move back to PostController.php we can see that it's hitting the index and currently doing nothing let's change that now to see if its working.

```
public function index()
{
    return "it's working";
}
```

Now if we go to the address <http://127.0.0.1:8000/post> we will see the following:



Laravel Fundamentals – Controllers – Passing data

Lets go back. Now Remember that we can pass in data as well into the controller. But how would we do this now, well last time we demonstrated using curly brackets with some type of variable enclosed like demonstrated below:

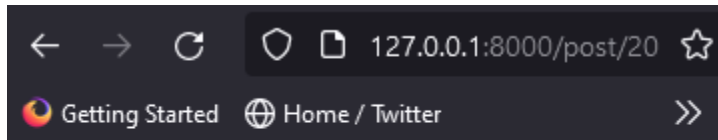
```
Route::get('/post/{id}', [PostController::class, 'index']);
```

After we pass the variable we move back to the PostController.php file and declare it in the index() method and use it as normal:

```
public function index($id)
```

```
{  
    return "it's working" . "The number " . $id;  
}
```

We should receive an output of:



it's workingThe number 20

Let's go back to our web.php file. AS you can see here you might be wondering how does it know that it has to pass the id? Don't we need a function? Well we already have a function. If you go to post controller we have the index function. It is going to take everything from the url. So we could keep taking variables if we wanted to. So that's we pass information. That's one of the ways we can pass information. We can also pass on information (remember that we can go back to routes) and do the closure function we previously demonstrated and return the ID from here in that closure function from routes. In this next section, we will demonstrate how to make a special type of controller. A controller that uses a resource, as well as a route with a resource.

Laravel Fundamentals – Controllers – Resources and Controllers

What is a resource controller? A resource controller is a type of function, A static function like get. That will give us access (automatically) to different types of routes that we can use in our controller and ill show you right now how that works. Back in the web.php file comment out our previous route and write the following method

```
Route::resource();
```

The resource function will help us very simply by creating special routes for us that we can use in the controller. For example.

```
Route::resource('posts', \App\Http\Controllers\PostController::class);
```

I don't have to type in the method anymore because I'm not just sending requests for the method index, I am sending a request for Create, Update, delete, etc. So let me show you right now. So if I go back to bash, and I type in – php artisan route:list you should receive the following output:

POST	_ignition/execute-solution ..	ignition.executeSolution	> Spatie\LaravelIgnition > ExecuteSolutionController
GET HEAD	_ignition/health-check	ignition.healthCheck	> Spatie\LaravelIgnition > HealthCheckController
POST	_ignition/update-config	ignition.updateConfig	> Spatie\LaravelIgnition > UpdateConfigController
GET HEAD	api/user		
GET HEAD	posts	posts.index	> PostController@index
POST	posts	posts.store	> PostController@store
GET HEAD	posts/create	posts.create	> PostController@create
GET HEAD	posts/{post}	posts.show	> PostController@show
PUT PATCH	posts/{post}	posts.update	> PostController@update
DELETE	posts/{post}	posts.destroy	> PostController@destroy
GET HEAD	posts/{post}/edit	posts.edit	> PostController@edit
GET HEAD	sanctum/csrf-cookie	sanctum.csrf-cookie	> Laravel\Sanctum > CsrfCookieController@show

With that route I gave in the first parameter of the method post, it created for me some additional routes.

- Posts.index
- Posts.store
- Posts.create
- Posts.update
- Posts.destroy
- Posts.edit

So it created all of the above resources for me automatically, what that does for me is that not only does it create the names but it also creates the methods in which there are sent. So we can see that this post we created is sent as a get request.

All right?

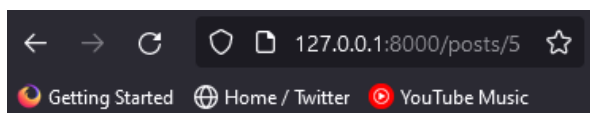
Because we don't have any information we aren't sending data, but for the posts.destroy we are sending some type of ID then we are deleting it. Another example is posts.update to update we send the id as a put or a patch, but we will take about this a bit later in the lecture on how to use a patch.

So lets demonstrate this now.

If I wanted to see something I would use posts.show. Now let's go back to the PostController.php and specifically to the show method and return some text to text if its working as follows:

```
public function show(string $id)
{
    //
    return "This is the show method" . $id;
}
```

Now when we go to the address <http://127.0.0.1:8000/posts/5> we should see the following:

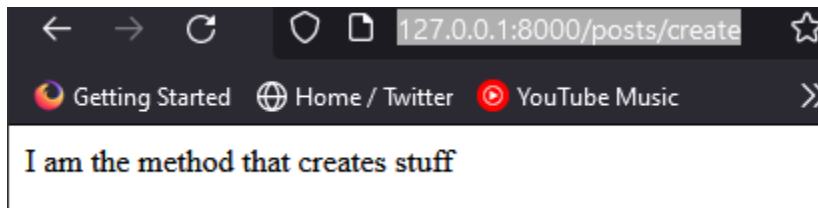


This is the show method5

Now if we want to access another method let's say create we can return as follows:]

```
public function create()  
{  
    return "I am the method that creates stuff";  
}
```

So now if I go to the url <http://127.0.0.1:8000/posts/create> . I should receive the following:



So what we did was using the resource, we were able to create all of the previous route parameters for us automatically with our methods that the data gets sent as well as name routes. We could use the names routes in whatever we want in our application. We can use it in a link, for ex if the user goes to post.index then goes to this portion of the website, which in turn allows the user to post, create an account, or whatever functionality your website contains. So to summarize, with resource, we get all of the above actions, the name routes, the methods for sending the data, and the URLs. So if the user goes to post, we will automatically create a page that will send the post request to the database to ask for information or to input information securely.

All right?

We're going to see that in the application that we will be building later on.

I want you to have this information so you can fully utilize resource, and how we communicate with our post-controller methods. You can go ahead and display some text and it will work in a similar way, as long as you follow this convention. If you don't follow the convention then it's not going to work, and make sure if the controller method you use has a parameter that you are passing a parameter into it.

(Routing documentation: <https://laravel.com/docs/5.2/routing>)

Laravel Fundamentals - Views – Creating views and custom methods

In this portion of the lecture, we will be taking a look at views. This is the last subject of the MVC design paradigm (modal, view, controller). We're going to be dealing with models a lot when we're building applications.

Where are view located?

Located in the resources folder, inside should be three folders css, js, and views and this is where our views are. Now let's use this resource that we created in the previous section below to display views:

```
Route::resource('posts', \App\Http\Controllers\PostController::class);
```

So let's go to the Http folder and open the PostController.php file. So let's say we're showing the user a contact page, to do this let's create a custom controller that returns a view as follows:

```
public function contact()
{
    return view('contact');
}
```

Now let's create our view. To do this create a file in the views folder named contact.blade.php (Instructors Note: don't worry about blade for now as this will be covered in the next section).

The reason why we append .blade. to our file is because Laravel uses a templating engine, or a special type of program to make developing with php easier for example instead of using this following syntax:

```
<?php echo $this ?>
```

We can eliminate all of this and be able to output variables like this

```
{{ $lksajdlaksjd }}
```

To get started with our contact page we're going to use the older php welcome page as a template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Contact me</title>

    <link href="https://fonts.googleapis.com/css?family=Lato:100"
rel="stylesheet" type="text/css">

  </head>
  <body>
    <div
      class="container">

      <h1>Contact page</h1>

    </div>
  </body>
</html>
```


Now this html does not contain any styles yet, but we will soon remedy that by installing bootstrap in a later section. First let's move back to our web.php file and create a route for the contact page like so:

```
Route::get('/contact', [PostController::class, 'contact']);
```

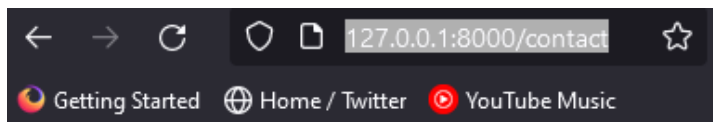
Back in the PostController.php, Now the contact function will look for a file named contact inside the views folder. But if you had another folder let's say for example we had our file within a separate folder named pages in the resources folder (or any folder other than views). You would have to change the controller method like the following:

```
public function contact()
{
    return view('pages/contact');
}
```

(Instructors Note: Go ahead and remove the pages/ from the controller method)

Now let's test our contact view by going to the following url <http://127.0.0.1:8000/contact>

And it should display the following output:



Contact page

Laravel Fundamentals - Views – Passing data to views

Now that we know how to create view now its time for us to learn how to pass data into views. First as always the super simple approach and then a proper configuration with routes. First in the web.php file we're going to create a custom method, and then we're going to create a view as well. As shown below(for testing coment):

```
Route::get('post/{id}', [PostsController::class, 'show_post']);
```

All right even though we already have a show controller method in the resource, we're going to use our custom one called show_post. Now that we have our routes in place, lets move back to our PostController.php file and create the following method:

```
public function show_post(){
    return view('post');
}
```

So this function automatically will search for any files with the name post in the view folder. Now in the views folder create a file named post.blade.php now paste the following into the file to get started.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Contact me</title>

    <link href="https://fonts.googleapis.com/css?family=Lato:100"
rel="stylesheet" type="text/css">

  </head>
  <body>
    <div
      class="container">

      <h1>Post </h1>

    </div>
  </body>
</html>
```

So I talked a little bit about the blade engine, right?

Previously I mentioned that the blade engine is a templating engine that Laravel uses to execute php in a simplified way; instead of writing out php tags we can simply place curly brackets around our php code. Now lets move back to our Postcontroller.php file to make sure that we are passing the data correctly and modify the show_post accordantly (we're getting the \$id from web.php).

```
public function show_post($id){
    return view('post');
}
```

Now we're going to pass the id into the view, so there are a couple of ways of doing this. First we could 'chain' another term for this dependency injection the function, accepting two parameters the name of the variable and the variable itself like the so:

```
public function show_post($id){
    return view('post')->with('id',$id);
}
```

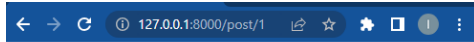
The above id variable is going to come from the url passed in from the get request, is now going to be available in the post.blade.php file. Why? Because the postcontroller.php view function is opening(injecting) the variable into the view. Now we're going to move to the post.blade.php and display it. Modify the container div like so:

```
<div
    class="container">

    <h1>Post {{$id}}</h1>

</div>
```

So now if we go to the following address <http://127.0.0.1:8000/post/1> I should receive the following output:



Post 1

Now there is a better way of actually doing this when you want to pass multiple parameters. We use a native function named compact that allows me to accept any number of parameters, but if I pass a parameter that has the same name as a variable (that I'm passing into the function) it will convert the name into the actual variable. Back in the PostController.php file make the following changes to the show_post function:

```
public function show_post($id){
    //return view('post')->with('id',$id);
    return view('post', compact('id'));
}
```

Why use this function over with dependency, well with the compact function I can pass multiple variables like so:

```
public function show_post($id,$name,$password){
    //return view('post')->with('id',$id);
    return view('post', compact('id'));
}
```

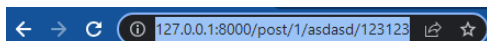
With modifications to the web.php file like so:

```
Route::get('/post/{id}/{name}/{password}/',[PostController::class, 'show_post']);
```

Now we add the parameters to the view like so:

```
<h1>Post {{$id}} {{$name}} {{$password}}</h1>
```

Now if we go to this address <http://127.0.0.1:8000/post/1/asdasd/123123> we get the following:



Post 1 asdasd 123123

Laravel Fundamentals - Laravel Blade templating engine

In this section of the lecture, we are going to be discussing blade. Without dragging out the details, essentially what Blade does is it grabs our php file and makes it easier to implement in HTML. For example, instead of writing out an echo statement, blade interprets two curly braces with a variable inside as a print statement. That's essentially what blade does, on top of that blade also includes a whole host of functions that are rudimentary to use.

Laravel Fundamentals - Master layout setup

In this portion of the lecture, we will dive deeper into blade. Blade is formally known as a templating engine. PHP has its own native PHP templating engine, but it lacks some modern features that make it difficult to coexist in the modern web development space. One said feature of blade is its ability to create a master template, which will be discussed now.

Back in the `contact.blade.php` file we can see the following contains extraneous HTML tags that can be streamlined with blade. For example, imagine if we wanted to have a different view for each navigable portion of our website: we would have to copy and paste just every HTML element excluding the body. If we take notes from our PHP routing we can see that when we want to reuse the same navigation over and over again we solve this problem with our keywords: `namespace` and `use`, we then break this down into separate header and footer files, etc. Well, these are all features that can be done in standard PHP, but in Laravel we can improve on this.

First start by creating a new file directory in the views folder. Title it `layouts` and create a file named `app.blade.php`. (Note that this is a standard naming convention for Laravel developers) Next delete everything in the file and enter a `!` and press tab. The following HTML template code should be present:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

Next insert the following div into the body tag (this is done so we can incorporate bootstrap further on) now we insert the blade functions `yield` in the container class and below the container. The first function call would accept the parameter `content`, and the second function should accept the parameter `footer`. It should now look like the following:

```
<body>
  <div class="container">
    @yield('content')
  </div>
  @yield('footer')
</body>
```

Now let's move to the contact.blade.php file, first remove all of the code from the file, next at the top of the page we call the extends function passing that accepts the parameter layouts.app (note that we do not have to type the .blade.php to call this file) the file should appear as follows:

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

This extends function includes everything contained in the app.blade.php file. Now in order for our content yield, we must use a function called section and the function @endsection. The section function accepts the parameter content. Now we can Type the following into the contact.blade.php file.

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

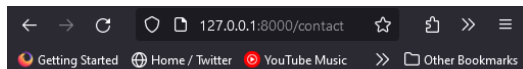
@section('content')
  <h1>Contact Page</h1>
@endsection
```

Now remove all of the code from the post page and copy paste the following:

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

@section('content')
  <h1>Post Page</h1>
@endsection
```

So to test that our layout is functioning properly go to the url <http://127.0.0.1:8000/contact> and the following should be displayed:



Contact Page

To see how we are including the html template in our separate files, right-click the viewport and view the page source. It should appear as follows:

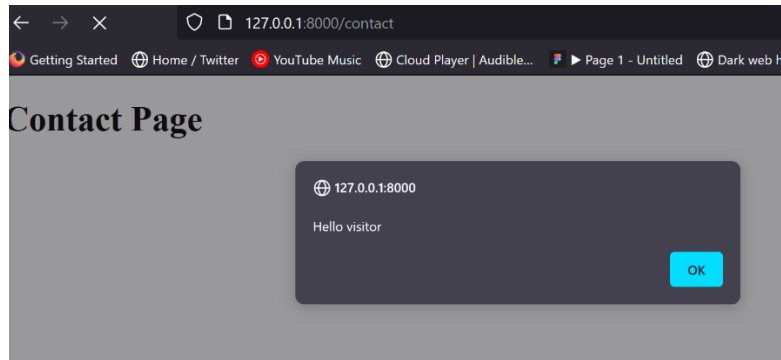
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
  <div class="container">
    <h1>Contact Page</h1>
  </div>
</body>
</html>
```

The beauty of this solution is a reduction of boilerplate HTML code for every view while maintaining unique navigable views. For example, let's say in this contact page I want to display some javascript I would approach it like so (this code goes below the content section):

```
@section('footer')
  <script> alert('Hello visitor')</script>
@endsection
```

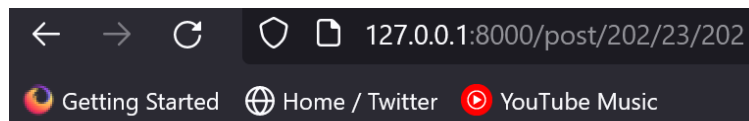
Now when we save and refresh <http://127.0.0.1:8000/contact> we should receive the following alert message:



Move back to the post.blade.php file and add the following variables to fulfill the post parameters:

```
<h1>Post Page {{$id}},{{$name}},{{$password}}</h1>
```

When we go to the following url we should receive this response:



Post Page 202,23,202

As you can see with blade templates and this layout, we can add only the content we have specific to add on specific pages, with the least amount of code reuse. In the next section, we will go over a few more examples with blade.

Laravel Fundamentals - more blade features

So far we have seen a rather impressive developer experience with Laravel, alas this is only the tip of the iceberg. In our next topic of discussion, we will be passing data into our blade views and processing it. So let's say for example we are in the PostsController.php file and declare an array named People and add a function call to the contact return statement with the people array as calls parameter, its should look something like this:

```
public function contact()
{
    $people = ['Luka', 'Jose', 'James', 'Peter', 'Maria'];
    return view('contact', compact('people'));
}
```

Now on the contact.blade.php page (note This is not how we would actually do this, But it is a mere sample of blades capabilities) say we want to use an if statement in blade. Now as you've probably noticed most of what you do in blade uses the @ symbol, so for the if/all blade control structures or looping statements there is no exception. Below we are passing the native PHP function called count into the if statement, and we are going to count the number of elements in said array. It should look something like this (meaning that if the if statement reads a positive integer it activates):

```
@section('content')
    <h1>Contact Page</h1>

    @if (count($people))

        @foreach ($people as $person)
            <li>{{$person}} </li>

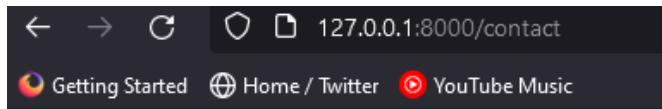
        @endforeach

    @endif

@endsection
```

Also, go ahead and comment out are alert in the footer for now, Note to quickly comment out a chunk of code highlight it hold ctrl+/.

It should output the following at the url <http://127.0.0.1:8000/contact>:



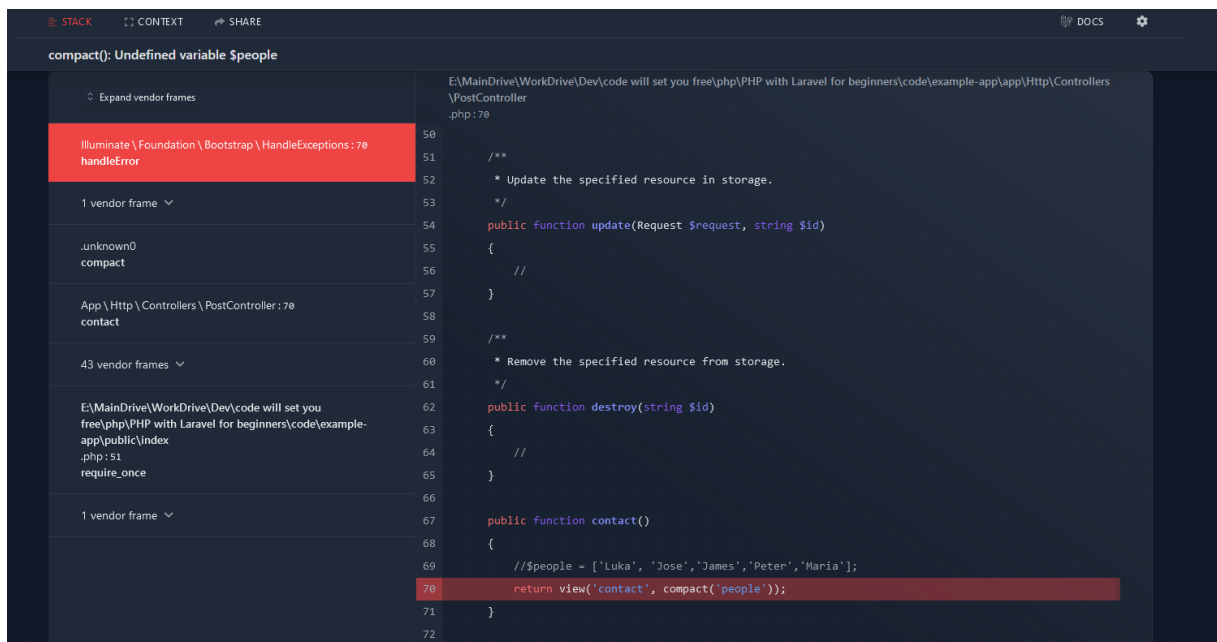
Contact Page

- Luka
- Jose
- James
- Peter
- Maria

As you can see we are echoing everything out. Let's now move to the PostController.php and comment everything out to see what happens, like so:

```
public function contact()
{
    //$people = ['Luka', 'Jose', 'James', 'Peter', 'Maria'];
    return view('contact', compact('people'));
}
```

What do you think is going to happen when we go to the URL at <http://127.0.0.1:8000/contact> ? Let us see:

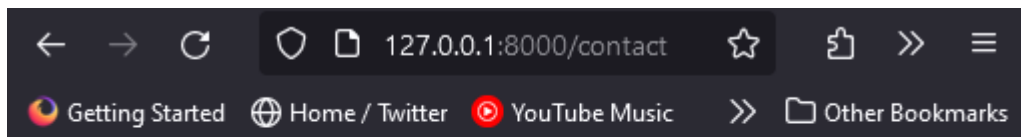


In the fancy-looking debug screen we see that our error is 'compact(): Undefined variable \$people' in line 70. Let's try another variation. This time let's declare the \$people array, but store nothing in it like so:


```
public function contact()
{
    //$people = ['Luka', 'Jose', 'James', 'Peter', 'Maria'];

    $people=[];
    return view('contact', compact('people'));
}
```

Now lets check what happens when we refresh the url <http://127.0.0.1:8000/contact> :



Contact Page

As mentioned prior, our if statement accepts an argument of 0 meaning that It does not execute. So I hope you see how useful our blade templating engine is, because we are going to be using this a lot when we are creating our application so make sure you get this down before I leave this topic I want to show you this blade function:

```
@include('')
```

Just as we used before with our native php include statements, we can include files within blade views. So, to close, think about this: What use cases would you to include files in views, well as it turns out most dynamic algorithms are written with inputs that are subject to change. Be it information queried from our database, a separate class object can tell a php controller to gather information from itself, and store it into a custom file solution, then our blade views can parse the file and use it in some sort of visualization algorithm. I'll be frank with you seeing this for the first time was incredibly magical for me, finally a framework that neatly partitions and streamlines all server-side rendering.

[Blade Templates - Laravel - The PHP Framework For Web Artisans](#)

Laravel Fundamentals – Database – Laravel Migrations

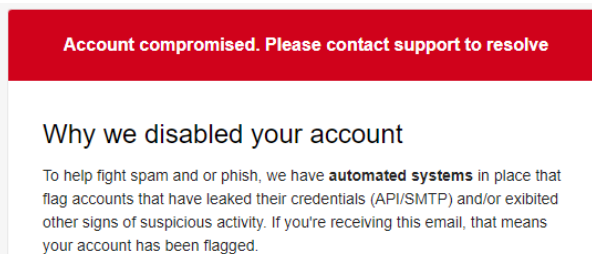
It is time for us to learn about migrations, migrations are easily one of the best features Laravel has to offer. In a nutshell, migrations are a way to generate tables and the columns that are contained in said tables. Basically, we define the table and columns inside our application using Laravel, then we run a PHP artisan command to create the table in whatever database software you want ex. (SQL, MySQL, SQLite, etc) for this course we are using MySQL, so you would be able to see that table in phpMyAdmin (phpMyAdmin is a free software tool written in PHP, intended to handle the administration of MySQL over the Web.), or whatever other program you use to view your tables. If we are using SQLite we can also use the same method to create a file-based database that we could also view. Why is this so important to use migrations? Imagine you have to share your project, you no longer need to use

phpMyAdmin to export the database to a CSV file and then create that database in their domain language. You can just run the php artisan command then voila, you have your tables created automatically on their side.

Laravel Fundamentals – Database – Laravel Migrations – Environment Configurations

In this very important portion of the course, we are going to be talking about Laravel's configuration files and environment variables, right now navigate to `.env.example`. As you can see from our project we have two files the environment variables (`.env.example`), and the configuration file (`.env`). Now when you are using composer, composer will automatically generate the two files. If you are not using Composer, and you are somehow doing this manually, you will only generate the `.env.example` file. Why? Well because you never, under any circumstance want to push a project into production with personal information. If you navigate to the `.env` file you will see variables like `AWS_SECRET_ACCESS_KEY`, `MAIL_PASSWORD`, and `DB_PASSWORD`. There are a lot of things in this that are personal to your specific application, so our solution to this is we place the actual sensitive information in the `.env.example` file, things that make your application function, everything else should be left as default. Let's say, for example, you pass your project to another developer and tell them that you're finished with the application and it's your job to keep developing this you are going to send it with `.env.example` file and they will have to rename it to `.env` to make it work. This is very important, so remember this when working with other students' code so that's why we have a two-file system for environment variables. Now, a quick personal aside, In one of the hack-a-thons I was competing at I was given a `.csv` file and a Python visualizer for geospatial map data of Dallas. As the project manager, I was tasked with working with a front-end developer to create a website and a database for the `.csv` file. As the day progressed, I was working at a reduced pace as my team keep asking me about potential features we could add to our application, this is known as feature creep and admittedly I was hungry to impress not only judges by my teammates, so I agreed. Given that I only had 24 hours to create this I got sloppy and didn't fully understand the tools I was using when I did the unthinkable... I pushed code to a public repository that contained my full mail-gun API key for the entire world to see. Well, one might think, sure there are plenty of GitHub projects out surely nothing bad would happen, plus I wasn't even deploying this project so only my future hiring manager would see it. I was dead wrong, in a matter of moments after pushing that code I received an e-mail from mail gun,

it read as follows:



So I implore you, my dear students, to learn from my mistakes and understand the importance of properly configuring your environment variables.

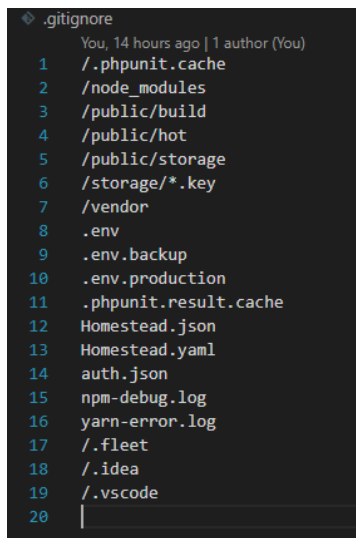
Back to the lecture material:

Now later in this course if you hear me saying constants, or deployment keys and values, just know the formal name for it is environment variables. The `.env` file is an environment file, and for those who are ready and have a firm understanding of GitHub the `.env` file is automatically excluded from version control. And for students that don't, GitHub is a code repository where developers check out fellow

developers' work and demonstrate their own. For instance, you might find a project that interests you and think, you know what, this could really use a graphical interface. Well, you can download the project and run it on your machine and make changes that are independent of the original source code. If you think you have contributed enough to the project, enough to make the user experience smoother, or perhaps a feature that expands the project without removing its original functionality, you can then submit a merge request to the source code author and see if it gets accepted. Now you might receive a flat-out no with little reason why, you might receive some feedback that certain parts need to be tweaked for you to resubmit it, or it might get flatly accepted. When it does get accepted you can now proudly brag to your significant other or web development instructor that you help make so and so project possible.

At the very least I would be impressed, and hey so might the person that's trying to hire you, especially if that hiring manager works for the open-source project you are working on and has funding / paid developers. If your contributions/features stand out in code reviews, against developers that are paid to have a deep understanding of the project, that hiring manager would be foolish not to hire you.

As mentioned previously the file `.env.example` will be automatically ignored by Git Hub when you push your project, but how? If you navigate to the `.gitignore` file you will find a list of files that GitHub will ignore when you push your project to a GitHub repository with Git take a look:



```
.gitignore
You, 14 hours ago | 1 author (You)
1 /.phpunit.cache
2 /node_modules
3 /public/build
4 /public/hot
5 /public/storage
6 /storage/*.key
7 /vendor
8 .env
9 .env.backup
10 .env.production
11 .phpunit.result.cache
12 Homestead.json
13 Homestead.yaml
14 auth.json
15 npm-debug.log
16 yarn-error.log
17 /.fleet
18 /.idea
19 /.vscode
20 |
```

The bottom line is never place any sensitive data in any file other than files ignored by git, and if you are not using source control, make sure you remove the `.env` file and document everything that a user needs to run your code. Now that we know how to use environment variables safely, let us demonstrate how we use them in our project.

Navigate to the config folder and open the `database.php` file.

As you can see the `database.php` is simply returning a two dimensional array filled with a plethora of keys and values. But let us explain a bit of what's going on here:

```
'connections' => [  
  
    'sqlite' => [  
        'driver' => 'sqlite',  
        'url' => env('DATABASE_URL'),  
        'database' => env('DB_DATABASE', database_path('database.sqlite')),  
        'prefix' => '',  
        'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),  
    ],  
],
```

(in the code above) We are using not only keys and values but also use a helper function, `database_path()`, to help define our database path.

```
'mysql' => [  
  
    'driver' => 'mysql',  
    'url' => env('DATABASE_URL'),  
    'host' => env('DB_HOST', '127.0.0.1'),  
    'port' => env('DB_PORT', '3306'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'unix_socket' => env('DB_SOCKET', ''),  
    'charset' => 'utf8mb4',  
    'collation' => 'utf8mb4_unicode_ci',  
    'prefix' => '',  
    'prefix_indexes' => true,  
    'strict' => true,  
    'engine' => null,  
    'options' => extension_loaded('pdo_mysql') ? array_filter([  
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),  
    ]) : [],  
],
```

In this example, we use the `env` helper function and pass in two parameters: the key's name and its value. For the database variable, we pass in the name of our key and the string `forge`. What `forge` does it retrieve the environment variable from the `.env` file? The nice thing about Composer is that this is taken care of automatically, you can also set this up manually if so desired.

So you might be asking, what's the point of the `.env` file? Why don't we directly call the parameters from the `.env.example` file? You might think the goal is to ensure sensitive information is not directly used but instead parameterized. But we still run the risk of exposing personal information, thus, we set the data to auxiliary variables.

So back to the `database.php` file, we see that name of our array is `connections` but contains what are called drivers. We have a driver for the following database tools: SQLite, MySQL, PostgreSQL, and SQLServer.

So for those that don't know SQLite, a brief description, SQLite is a file-based database written in C. Meaning we do not have to use a specialized program to create and view the database, it's all saved into a single file. So if we have a small application that does not require the overhead of MySQL, we could simply use SQLite and save it to a file. Think small or Embedded systems. As you can see from the previous SQLite driver code we are calling the helper function `database_path` to create the file `database.sqlite` to store all of the information in the database. For example, users, posts, or categories are all going into the file. Of course, as you are probably aware in this course we use MySQL we will still cover MySQL to make sure everyone is up to speed on this in a later section, and we will cover how to use SQLite in an even further section so look forward to it.

Anyways, the config file also contains similar PHP files for environment variables, and just like our `database.php` file, it gathers this information from our secure `.env` file.

Now if you want more details on the `.env` file I recommend reading the following GitHub repository <https://github.com/vlucas/phpdotenv>. This expands on what I've mentioned so far, including the origins of the file (spoiler it's Ruby on Rails), and the architecture behind it all, etc. Very cool stuff!

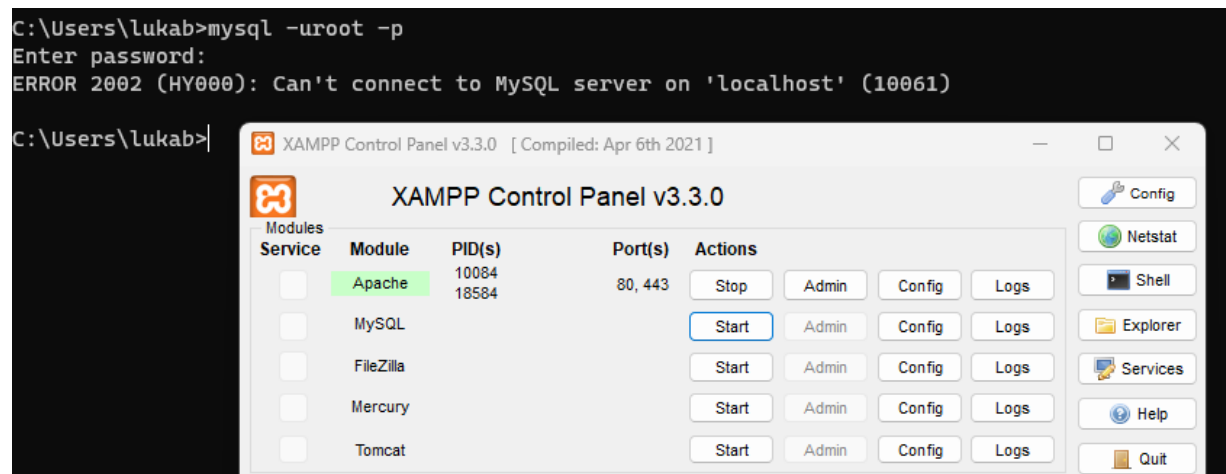
Laravel Fundamentals – Database – Windows OS - Migrations

This portion of the course is for Windows users, if you are using the Mac operating system we will cover that in the next section of this course.

So the time has come to demonstrate one of the most powerful features of Laravel, Migrations. We'll start off with making sure everyone is on the same page and has properly configured MySQL, next a general overview of Migrations, as it is truly one of the most important features of Laravel. The concept, as mentioned in the previous section, is not that difficult to grasp.

All the way back in section two I demonstrated how we configure and access through the command line and through the web browser MySQL, by using the CMD/command line command `mysql -uroot -p`, and with PHPMyAdmin, right? So as a refresher open up the command line and type the following command

`mysql -uroot -p`. Now unless you have already opened XAMPP and have started the MySQL service you would receive the following error:



Now If you haven't already opened up XAMPP and started the Apache and MySQL service, note the Apache service allows us to access php myadmin. Back in CMD press the up arrow to return the previous command we entered, It should look something like this:

```
C:\Users\lukab>mysql -uroot -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.4.28-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Now to display all of our databases we use the following command show databases; . Next type the command cls (note: if you get stuck and need to close out of your current session, for example you forget the semi-colon you can type the command CTRL+c). if your command line displays as follows you know you have typed it in correctly:

```
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| my_db         |
| mysql         |
| performance_schema |
| phpmyadmin    |
| test         |
+-----+
6 rows in set (0.052 sec)

MariaDB [(none)]> cls
->
```

Next we are going to create a new database, go ahead and type CTRL+c to restart the session, enter mysql -uroot -p and then enter the following command: create database new_cms; If you have typed everything correctly it should output the following:

```
MariaDB [(none)]> create database new_cms;
Query OK, 1 row affected (0.008 sec)

MariaDB [(none)]>
```

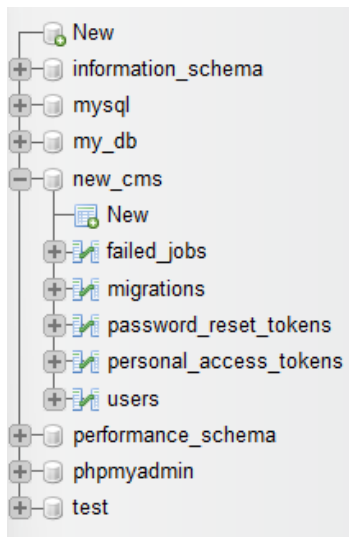
So once you have created a new database go to the following link <http://localhost/phpmyadmin/index.php> and you will see the new database. Once you have created a database you no longer need to do anything else in Laravel to configure the server. It takes care of the

creating of the tables and their columns and that's is done with migrations. (Instructor only)In the new_cms open the .env file and lets make sure that are Laravel is actually connected. So in the .env file we enter the value for the variables DB_DATABASE = new_cms. The variable DB_USERNAME = root should be left as default (for not) and the the variable DB_PASSWORD = should be keep blank (for now.)It should look like the following:



























```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=new_cms
DB_USERNAME=root
DB_PASSWORD=
```

Now once we are connected here make sure that you save it, now we can start to create columns in are bash terminal. First make sure that your terminal is inside your project, next enter the following - command php artisan migrate . You should receive the following back in the terminal letting you know what tables have been create.

Now if we go back to the following link <http://localhost/phpmyadmin/index.php> we should see the following 5 tables:



Now if we click on the data base named users we see that we have generated the following columns:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1 id 	bigint(20)		UNSIGNED	No	None		AUTO_INCREMENT	 Change  Drop  More
<input type="checkbox"/>	2 name	varchar(255)	utf8mb4_unicode_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	3 email 	varchar(255)	utf8mb4_unicode_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	4 email_verified_at	timestamp			Yes	NULL			 Change  Drop  More
<input type="checkbox"/>	5 password	varchar(255)	utf8mb4_unicode_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	6 remember_token	varchar(100)	utf8mb4_unicode_ci		Yes	NULL			 Change  Drop  More
<input type="checkbox"/>	7 created_at	timestamp			Yes	NULL			 Change  Drop  More
<input type="checkbox"/>	8 updated_at	timestamp			Yes	NULL			 Change  Drop  More

At this point, you might be asking your self, how is Laravel doing this? Well lets go back to the project and take a look at this. Navigate to the folder named database, open the migrations folder, and by default you should see four migrations, perhaps in a future Laravel update there will be more. If you click on the first file named ..._create_users_table.php you will find the following code:

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('users');
    }
}
```



```
};
```

As you see we define a class that we are inheriting from an anonymous parent class named migration. Hopefully, you recall how this works but if you don't here's a quick refresher. This PHP file is using inheritance. It defines an anonymous class that extends the Migration class provided by the Laravel framework. By extending the Migration class, this anonymous class inherits all the properties and methods defined in the parent class. We define two functions, up and down: up is the function that actually creates the table and columns. As you can see here are function accesses the Schema class and calls the Schema function to create. We pass in two parameters, the first parameter is the name of the table we want to create, and the next parameter we pass is a closure function that has a blueprint object injected into it. Our \$table is an instance of the Blueprint that calls seven methods to create our table.

Each of the methods creates a column of the type relative to the name of the method. For example, in the first method call, we create a column of the type id, in the second method we create a column of the type string, and we name the column based on the parameter; meaning that the second column is called name and so on. Looking now at our phpMyAdmin we can see our columns and the variables that they hold. For example, we can see that in MySQL the string typed is defined as a varchar(255) with a buffer size of 255 characters. Now you may wonder why we have an extra column, well the column created_at and updated_at is created with one method: that method being timestamps(). Now column 6 remember_token accepts a varchar(100) with a buffer of 100 characters. So the key takeaway is the methods discussed are defined by the object we are passing as a parameter, this is the up function.

Now the down function essentially destroys the table. Now you can manage these functionalities with php artisan. If you enter the command php artisan without any other command and scroll down to the field that starts with migrate you can see the following

```
migrate
migrate:fresh      Drop all tables and re-run all migrations
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
```

Some of the important ones are:

- Migrate: fresh removes all the tables and re-runs the migrations over again
- Migrate: refresh resets and re-runs all of the tables

For now, we're not going into too much about what exactly these mean, but we will certainly be using them in a future portion of the lectures, so just be aware.

To summarize. Laravel migrations take care of the creation of tables so we no longer have to manually have to create them. Still, every time you want to add a column to a table you have to add it to the function call stack. For example, if I create another table using a schema call and name it shoes it would look something like this:

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

```
$table->string('Brand');  
});
```

This is certainly a lot easier to create as supposed to creating it in phpMyAdmin and a whole lot faster than creating it manually with the MySQL syntax, Laravel simplifies greatly.

One more thing, I should point out there are many more column types that are available in the Laravel documentation <https://laravel.com/docs/10.x/migrations#columns>

Laravel Fundamentals – Database –MAC OS – Migrations **//TODO**

Laravel Fundamentals – Database – Creating and dropping Migrations

In this portion of the lecture, we will discuss more about migrations. Now that we know how migrations work and how to create them, let's demonstrate how to build a migration from scratch.

Now if you don't remember from last time, navigate to the database folder, and inside this folder is another one named migrations, and we have our migrations.

Next, open a separate bash terminal from the one that is currently running the serve command, and lets create a migration. How do we do this? If you remember from the last portion of the lecture