**Laravel Installation and Configaration**

Download then setup to **xampp>php>php.exe** Laravel installation as follows:

**composer global require "laravel/installer"**

Now we have to search laravel then ctrl+shift+right click and open command window and write **php artisan serve** for server start.

**Routing & Controller in Laravel**

**Routing** হচ্ছে URL; কোন পেজ view/request করার জন্য কোন নাম define করে দেওয়াই হচ্ছে Routing এর কাজ। i.e. localhost/**contact**, localhost/**about** etc.

routes/web.php page এ যে default GET method টা আছে এটি দুটি parameter নেয়, একটি **string** and **anonymous function (Callback)** এবং **View** থাকে হচ্ছে Resource folderএর ভিতর এবং **Controller** থাকে হচ্ছে App/HTTP/Controllers.

View এর প্রতিটি পেজকে আমরা routes/web.php থেকে Call করতে পারি কিন্তু এটি কোন সঠিক পদ্ধতি না; সেজন্য আমরা View এর পেজ সমূহকে Controller (controller@method) এর মাধ্যমে Call করব। Controller তৈরি করার জন্য Command হচ্ছে:

**php artisan make:controller page\_name**

**page, css, script, image dynamic**

**এজন্য ২টি কাজ করতে হয়**:

১. Page গুলোকে পোজেক্ট ফোল্ডারে রাখতে হয়

২. CSS ও অন্যান্য Assets গুলোকে Public ফোল্ডারে রাখতে হয়।

**তারপর,**

HTML::script() is used for .js files and HTML::style() is used for css.

For Example:

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

{{ Html::script( 'js/app.js') }}

**OR,**

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

<link href="{{ asset('css/masterCss.css') }}" rel="stylesheet">

<link href="/css/master.css" rel="stylesheet">

**N.B**, “!!” comes from composer package collective/html এবং Default package illuminate/html

1. **Laravel Collective Package**

Laravel Collective is a third party package. It provides us dynamic front end utility specially form element.

For adding Laravelcollective to our application:

**composer require laravelcollective/html**

In your composer.json file at "require" group should be added below line:

**"laravelcollective/html": "5.4.\*"**

Then update composer with below command:

**composer update**

In config/app.php add following lines:

at providers group:

Collective\Html\HtmlServiceProvider::class,

in aliases group:

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

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

Now we can dynamic our Link, CSS, Images, Script as follows:

**{!! Html::style('assets/bootstrap/css/bootstrap.css')!!}**

**N.B**, For getting update tips/documentation go to laravelcollective.com

**Making Migration**

Database এর সাথে Connection:

.env ফাইলটাই Database Connection ফাইল।

In command line:

**php artisan make:migration create\_users\_table --create=users**

তাহলে user migration তৈরি হবে। Migration is database scheme.

Table তৈরির জন্য following command run করতে হবে তা না হলে Table তৈরি হবে না

**php artisan migrate**

এবং Rollback বা আগের অবস্থায় ফিরে আসার জন্য বা এক স্টেপ পিছেনে আশার জন্য

Command line:

**php artisan migrate:rollback**

To Rename table name and add new fields

php artisan make:migration rename\_posts\_to\_articles

**At up:**

Schema::rename('posts','articles');

**At down:**

Schema::rename('articles','posts');

**For adding new fields or modifying:**

php artisan make:migration add\_live\_and\_post\_on\_fields\_to\_the\_articles\_table --table=articles

**At up:**

$table->boolean('live')->default(0);

$tabli->timestamp('post\_on')->nullable;

**At down:**

$table->dropColumn(['live', 'post\_on']);

**Form Making**

1. **Restful Resource Controller:**

Restful Resource Controller তৈরি করলে Route.php পেইজ এ Route::get এর পরিবর্তে Route::resource ব্যবহার করে Controller টিকে include করতে হবে যাতে সহজে Image বা অন্যান্য ফাইল নিয়ে কাজ করা যায় এবং ফাংশনটা হবে নিচের মত:

Route::resource('registration','dbController');//Only Controller Load হবে, কোন ফাংশন হবে না।

1. {{ Form::open(array('route' => 'registration.store')) }}

//Html Form

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

এটার মাধ্যমে URL এ ডাটা পাঠানোর জন্য **page\_name.store** ('registration.store')

অথবা নরমালভাবেও পাঠানো যায় এক্ষেত্রে ফর্ম এর নিচে {{Form::token()}} এটা দিতে হবে।

1. **Accessing The Request** or **Input request** (Form Data controller থেকে দেখা):

public function store(Request $request){

$input=$request->all();

return $input;

}

1. **Connection between Controller and Model (Controller to Model):**

public function store(Request $request)

{

$input=$request->all();

//return $input;

myFirstModel::create($input);

return redirect('success');

}

url:localhost:8000/registration/create

1. **Model to Controller (Retrieving Multiple Models):**

public function index()

{

$allDBdata = myFirstModel::all();

return $allDBdata;

}

url:localhost:8000/registration

1. **Displaying Results In A View (Controller to View):** compact or with **ফাংশনের মাধ্যমে পাঠাতে হবে**

public function index()

{

$allDBdata = myFirstModel::paginate(1);

return view('dbTest.index', compact('allDBdata'));

//return $allDBdata;

}

url:localhost:8000/registration

**Model in Laravel**

The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

**N.B,** If table’s name is Articles (plural) then we should create a model named Article (singular) then the model will communicate automatically if other then we have to inform the name of the table to the model. It will also help in model relationship.

Model তৈরির জন্য:

**php artisan make:model ArticleModel**

**N.B,** Mass Assignmentisfor getting data to model from controller for storing to database.

Model Class এ নিচের Code লিখতে হবে:

protected $table = '$users';

protected $primaryKey='$user\_id';

protected $fillable = ['id', 'password'];

**Eloquent ORM & Query Builder**

Eloquent ORM is slower than Query Builder but we can make relationship with Models and easier to build & less code on the other side the Query Builder is a little bit hard but more faster.

**Eloquent ORM**

**Insert**:

//Method 01

$article = new Article;

$article->user\_id = Auth::user()->id;

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

$article->live = (boolean)$request->live;

$article->post\_on = $request->post\_on;

$article->save();

//Method 02

Article::create($request->all());

At Model:

protected $fillable = [

'user\_id', 'content', 'live', 'post\_on'

];

N.B, protected $guarded = ['']; or protected $fillable [];

public function setLiveAttribute($value){

$this->attributes['live'] = (boolean)($value);

}

//Method 03

Article::create([

'user\_id' => Auth::user()->id,

'content' =>$request->content,

'live' => $request->live,

'post\_on' =>$request->post\_on

]);

**Select**:

//All Records

Method 01:

$articles = Article::all();

return view('articles.index', compact('articles'));

Method 02:

$articles = Article::whereLive(1)->get();

$return articles;

//Paginated Records

$articles = Article::paginate(10);

return view('articles.index', compact('articles'));

//With soft deleted records

$articles = Article::withTrashed()->paginate(10);

return view('articles.index', compact('articles'));

//Only soft deleted records

$articles = Article::onlyTrashed()->paginate(10);

return view('articles.index', compact('articles'));

**Update**:

**Delete**:

//Soft deleting

$article = Article::findOfFail($id);

$article->delete();

return redirect('/articles');

**N.B**, For soft deleting we need an extra field to our table called **deleted\_at.** Hence we have to go to the model and import the class like this: **use Illuminate\Database\Eloquent\SoftDeletes;** and finally we have to add an attribute like this: **protected $dates = ['post\_on','deleted\_at'];** And then we have to go to the migration file and add like this: **$table->softDeletes();** and we have to refresh like this: **php artisan migrate:refresh**

//Without retrieving model

Article::destroy($id);

return redirect('/articles');

//For Multiple Deleting

Article::destroy([1,2,3,4]);

return redirect('/articles');

//Force deleting or Hard deleting

$article = Article::findOrFail($id);

$article->forceDelete();

return redirect('/articles');

**Query Builder**

**Insert**:

Method 01:

DB::table('articles')->insert($request->all());

Method 02: All but except \_token field

DB::table('articles')->insert($request->except('\_token'));

Method 03:

DB::table('articles')→insert([

'user\_id' => Auth::user()->id,

'content' =>$request->content,

'live' => $request->live,

'post\_on' =>$request->post\_on

]);

**N.B,** Query Builder don’t support accessor and mutator.

**Select**:

Method 01:

$articles = DB::table('articles')->get();

dd($article);

Method 02:

$articles = DB::table('articles')->whereLive(1)->get();

dd($article);

Method 03: Only one article return

$article = Article::table('articles')->whereLive(1)->first();

dd($article);

**Update**:

**Delete**:

**Eloquent Accessors & Mutators OR Custom Attributes**

The accessor and mutator both are public function in Laravel model for getting and setting model's attributes.

An accessor will automatically be called by Eloquent when attempting to retrieve the value of the first\_name attribute:

$fullName = $user->full\_name;

It's for customizing a model's attributes or adding fake attributes.

On the other hand mutator is for setting a real attribute of a model.

Mutator will be automatically called when we attempt to set the value of the an attribute.

1. Authentication or Login/Logout System

//

1. How to Register & Use Service Providers

**Introduction**:

Go ahead and look at the contents of the config/app.php file. You'll find an array entry that lists all the service providers that will be loaded during the bootstrapping of the Laravel application (with core aplication). If we want to load a dependency or service from service container (bucket) we need to load it via service provider. We can also create a custom service provider that will also be loaded automatically by laravel. So we need to inform laravel about the new service provider in config/app.php file at providers array.

**What Is a Service Container?**

In the simplest terms, we could say that the service container in Laravel is a box that holds various components, and they are served as needed throughout the application.

**For example:**

<?php

namespace App\Library\Services;

class DemoOne

{

public function doSomethingUseful()

{

return 'Output from DemoOne';

}

}

**What is a Service Provider?**

If the service container is something that allows you to define bindings and inject dependencies, then the service provider is the place where (register method) it happens that has already been discussed earlier in introduction.

**For example:**

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use App\Library\Services\DemoOne;

class EnvatoCustomServiceProvider extends ServiceProvider

{

public function boot()

{

}

public function register()

{

$this->app->bind('App\Library\Services\DemoOne', function ($app) {

return new DemoOne();

});

}

}

**How to Register Custom Service Provider**:

So you've created your custom service provider. That's great! Next, you need to inform Laravel about your custom service provider (in the file config/app.php) so that it can load it along with other service providers during bootstrapping.

**For example:**

/\*

\* Application Service Providers...

\*/

App\Providers\AppServiceProvider::class,

App\Providers\AuthServiceProvider::class,

App\Providers\BroadcastServiceProvider::class,

App\Providers\EventServiceProvider::class,

App\Providers\RouteServiceProvider::class,

App\Providers\EnvatoCustomServiceProvider::class,

**Register Method**:

To start with, we'll go through the register method to understand how you could actually use it. Open the service provider file app/Providers/EnvatoCustomServiceProvider.php that was created earlier and replace the existing code with the following.

**For example:**

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use App\Library\Services\DemoOne;

class EnvatoCustomServiceProvider extends ServiceProvider

{

public function boot()

{

}

public function register()

{

$this->app->bind('App\Library\Services\DemoOne', function ($app) {

return new DemoOne();

});

}

}

**Boot Method:**

The boot method which you could use to extend the core Laravel functionality. In this method, you could access all the services that were registered using the register method of the service provider. In most cases, you want to register your event listeners in this method, which will be triggered when something happens.

**For Example**:

public function boot()

{

View::share('key', 'value');

}

public function boot()

{

Validator::extend('my\_custom\_validator', function ($attribute, $value, $parameters, $validator) {

// validation logic goes here...

});

}

public function boot()

{

View::composer(

'demo', 'App\Http\ViewComposers\DemoComposer'

);

}

public function boot()

{

parent::boot();

Route::model('user', App\User::class);

}

**Dependency injection (uses):**

Here's the code somewhere in your controller where the dependency will be injected.

**For example:**

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Library\Services\DemoOne;

class TestController extends Controller

{

public function index(DemoOne $customServiceInstance)

{

echo $customServiceInstance->doSomethingUseful();

}

}

1. Model Factories

**Introduction**:

Laravel 5.1 comes with a feature called **Model Factories** that are designed to allow you to quickly build out “fake” models.

These have several use cases so that you can use them to build out “dummy” models which can be used for both seed data and in testing.

Normally we don’t need to create a new one, we can use the default UserFactory file for multiple factories.

Open database/factories/UserFactory.php and you will see a default one is already defined:

$factory->define(App\User::class, function (Faker\Generator $faker) {

return [

'name' => $faker->name,

'email' => $faker->email,

'password' => bcrypt(str\_random(10)),

'remember\_token' => str\_random(10),

];

});

$factory->define(App\Post::class, function (Faker\Generator $faker) {

return [

'subject' => $faker->sentence(5),

'description' => $faker->text(),

];

});

We are defining the “App\User::class” model as the first parameter and then a callback that defines the data that goes in the columns. This callback also injects **Faker** which is a PHP library that generates fake data. Faker is powerful and can be used for a number of different field types.

And as we can see the another factory is for the Post model in the same file. Hence we can make multiple factories.

If we create a new factory for a model then we need to write as follows.

Now, let’s create a new factory for our Post Model:

**php artisan make:factory PostFactory**

Here is the complete code for the factory:

$factory->define(App\Issues::class, function (Faker\Generator $faker) {

return [

'subject' => $faker->sentence(5),

'description' => $faker->text(),

];

});

The **--model** option may be used to indicate the name of the model for which the factory is created. This option will pre-fill the generated factory file with the given model:

**php artisan make:factory PostFactory --model=Post**

1. Database Seeding

**Introduction**:

Database seeds are a way of programmatically inserting **a number of data** into the database and one advantage to using them is you can quickly get dummy data into your app using seed classes. All seed classes are stored in the database/seeds directory. Seed classes may have any name you wish, but probably should follow some sensible convention, such as **UsersTableSeeder**

Normally we don’t need to create a new one, we can use the default DatabaseSeeder file for multiple seeders.

Now open the database/seeds/DatabaseSeeder.php file and adjust the run method to the following:

//Separately

public function run()

{

factory(App\User::class, 50)->create();

factory(App\Articles::class, 100)->create();

}

//Relationship

public function run()

{

factory(App\User::class, 50)->create()->each(function ($u) {

$u->posts()->save(factory(App\Post::class)->make());

});

}

Now before these seed classes are useful they need instructions on what to insert. Let’s use **model factories** for that. By default, the **db:seed** command runs the DatabaseSeeder class.

You may also seed your database using the **migrate:refresh** command, which will also rollback and re-run all of your migrations. This command is useful for completely re-building your database:

**php artisan migrate:refresh –seed**

If we want to create a new seeder then we have to follow the code:

Artisan Command:

**php artisan make:seeder UsersTableSeeder**

After creating the seeder, open it and paste it as follows:

public function run()

{

factory(App\User::class, 50)->create();

factory(App\Articles::class, 100)->create();

}

And then we can run them from DatabaseSeeder class as follows:

public function run()

{

$this->call([

UsersTableSeeder::class,

PostsTableSeeder::class,

CommentsTableSeeder::class,

]);

}

However, you may use the --**class** option to specify a specific seeder class to run individually:

**php artisan db:seed --class=UsersTableSeeder**

The easiest way is to use **tinker** artisan command without using above mentioned process is as follows:

**php artisan tinker**

And then write as follows for seeds:

**factory(App\User::class, 50)->create()**

Or we can use only for check only:

**factory(App\User::class, 50)->make()**

1. Difference between Model Factory and a DB Seeder

Database seeder is used to populate tables with data.

On the other side, Model factories (instructions) is a convenient centralized place to define how your models should be populated with fake data.

In seeder class you would leverage model factories, and model factories will most likely use another library to generate random fake data, such as **fzaninotto/faker**.

1. Database Testing