MVC

1. create the route in routes/web.php

eg Route::get(„/teams”, Maincontroller@teams)

1. in main controller create a public function

app/http/controllers/maincontroller

public function teams(){

return view(teams);

}

1. create a new view

@extends(„layout”);

name: teams.blade.php

content:

@section(„content”)

<h1>whatever</h1>

Így már működik az új teams oldalunk, de kattintással még nem lehet elérni. Hanem:

<a href=”{{ action (Maincontroller@teams) }}”></a>

Az action paramétere ugyanaz, mint a route

Elmenteni az új oldalt: resources->views-> user

projekt:

ha módosítottuk a less file-t, akkor a terminálba írjuk be, hogy „gulp” és az compile-olja nekünk.

In folders:

1. a)

create a new page with the content

eg: contact with the HTML inside

this is saved in a new folder under views

C:\coding\exercises\warmup-project\system\project\views\contact

b)

And in the same folder we create a new page\_layout.php file where we echo the content of the previous file:

page\_layout.php:

<?php

<div>

<?php echo $contact\_view; ?>

</div>

2.

we create a new controller =>

C:\coding\exercises\warmup-project\system\project\controllers

<?php

//contact

$contact\_view = new view('contact/contact'); // argument is the route

//page

$page\_layout = new view('contact/page\_layout');

$page\_layout->contact\_view = $contact\_view;

//sets the title of the page

presenter::setTitle('Contact');

//give the layout to the presenter

presenter::present($page\_layout);

megnézni, hogy működik-e:

<http://www.coding-exercises.local/warmup-project/?page=team>

a page utáni szó a controller neve!!

**Eloquent**

Különböző adat modellek vannak és mindegyik más nyelven beszél (MySQL, PySQL, Redis). Ezek RM – relation methods.

Hogy ne kelljen minden kérést háromszor megírni, Laravelnek van egy funkciója, az Eloquent. Ez ORM – object relation mapping. Ebbe egyféleképpen beküldjük a kérést és ő lefordítja bármelyik nyelvre.

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.

**To creat a new project in Laravel**

Itt van a részletes jegyzet:

<https://slavokozar.github.io/bootcamp/lectures/laravel/index.html>

Felmenni erre linkre, mert ott megtalálom az alábbi parancsot, amit a cmd-be kell írni:

<https://laravel.com/docs/5.4>

command line-on a megfelelő folderbe beírni. Nem kell már most blog foldert csinálnom, mert akkor kettő lesz belőle és ezt nem akarjuk

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

Az utolsó szó a projekt neve.

Ez megcsinál minden foldert és kapcsolatot helyettem.

**Defining Models**

The easiest way to create a model instance is using the make:model Artisan command:

php artisan make:model User

If you would like to generate a database migration when you generate the model, you may use the --migration or -m option:

php artisan make:model User --migration

php artisan make:model User -m

Bemegyünk a blog folderbe és ezt írjuk be, ha egy post modelt akarok (a model neve mindig egyesszám és nagybetű):

php artisan make:model Post

Így az app folderben lett egy post.php file.

Ez a tartalma:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

//

}

Az új modellünk, állhat egyedül is, ha mástól kapom hozzá az adatbázist.

De ha én csinálom meg az adatbázist is, akkor migrationra is szükség van, úgyhogy megint a blog folderbe beírjuk:

php artisan make:model Tag --migration

Szóval, csinálok egy adatbázist és az .env file-ban kell átírnom a dolgokat, hogy kapcsolódjon az adatbázishoz.

Megcsináltam a blog nevű tök üres adatbázist és az .env file-om második bekezdése most így néz ki:

DB\_CONNECTION=mysql

DB\_HOST=127.0.0.1

DB\_PORT=3306

DB\_DATABASE=blog

DB\_USERNAME=root

DB\_PASSWORD=

A többihez nem nyúlok.

A database/migration/új file neve alatt van a migrált file-om.

Ez most így néz ki:

<?php

use Illuminate\Support\Facades\Schema;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateTagsTable extends Migration

{

/\*\*

\* Run the migrations.

\*

\* @return void

\*/

public function up()

{

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

$table->increments('id');

$table->timestamps();

});

}

/\*\*

\* Reverse the migrations.

\*

\* @return void

\*/

public function down()

{

Schema::dropIfExists('tags');

}

}

A function up() megcsinálja a változtatásokat a táblázatban. A down() pedig visszacsinálja, ha mégsem kell vagy elrontottuk. Ez egyenlő a migration:rollback-kel.

Nekem a sárga rész a fontos, ide irkálok:

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

$table->increments('id');

$table->string("name");

$table->timestamps();

});

Ha van már egy Modelünk és nem csináltunk hozzá egyből migrationt, hanem utólag akarjuk, akkor

php artisan make:migration --create=posts create\_table\_posts

--create=posts (ez többesszamban kisbetűvel!) ez a neve a táblázatnak az adatbázisban

create\_table\_posts a neve az új file-omnak, ami úgy néz ki, mint a fenti idézett

Ha nem tudjuk pontosan ezeket a parancsokat, akkor kérhetünk segítséget:

php artisan make:migration --help

Ilyenkor a táblázatok még nincsenek benne a MyAdminer adatbázisban, nem kell idegeskedni. ☺

Csináljuk meg magát az adatbázist a MyAdminerben és utána

ezt írjuk be a command line-ba:

php artisan migrate

Ha a „way too long” errort kapunk -> a create passwords reset table migration-be menjünk be és ezt írjuk át

Schema::create('password\_resets', function (Blueprint $table) {

$table->string('email')->index();

$table->string('token')->index();

$table->timestamp('created\_at')->nullable();

});

erre

Schema::create('password\_resets', function (Blueprint $table) {

$table->increments("id");

$table->timestamp('created\_at')->nullable();

});

És próbáljuk újra.

Ha azt kapjuk, hogy user table alredy exist, akkor a create users table-ből (ez is migration) javítsuk ki:

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

$table->increments('id');

$table->string('name');

$table->string('email')->unique();

$table->string('password');

$table->rememberToken();

$table->timestamps();

});

public function up()

{

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

$table->increments('id');

$table->string('name');

$table->string('email');

$table->string('password');

$table->rememberToken();

$table->timestamps();

});

}

Ezután töröljük ki az adatbázisokat kézzel (a MyAdminerben!!) vagy a migrate:reset paranccsal és migráljuk újra (php artisan migrate).

## Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model.

<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {

echo $flight->name;

}

Ha csak kísérletezni akarunk, akkor írjuk, hogy

php artisan tinker és megkapjuk a shellt és játszhatunk.

pl kíváncsiak vagyunk az összes postra:

App\Post::all();

Most ez ugye üres, mert még nincs adatunk, úgyhogy rakjunk bele:

a tinker után írjuk:

$post=new App\Post(); => enter

utána berakom az adatokat

$post->name = „My first post”;

Ezt el is kell menteni az adatbázisba, ahhol, hogy az Adminerben megjelenjen.

De adjunk még neki adatokat.

$post->content = „content content content content”;

Itt az ideje elmenteni:

$post->save();

Ahogy megcsináltam, úgy módosíthatom is, amíg ugyanabban a tinkerben vagyok.

$post->name=”Other title”;

ne felejtsük újra elmenteni!

$post->save();

Most már van mit kiszedni az adatbázisból:

App\Post::all();

Ezt collectionnek hívják.

Ha nem mindent akarom, normál query buileder functions work:

$posts = App\Post::where('active', 1)

->orderBy('name', 'desc')

->take(10)

->get();

utána küldhetünk rá egysoros foreach-t

foreach($posts as $post) {echo $post->name . PHP\_EOL; }

PHP\_EOL miatt írja két sorba. Ezt nem muszáj belevenni, csak így jobban olvasható.

## Retrieving Single Models

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using find and first. Instead of returning a collection of models, these methods return a single model instance:

// Retrieve a model by its primary key...

$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...

$flight = App\Flight::where('active', 1)->first();

vagy

$post=App\Post::where(„id”, „>”, 5)->get();

You may also call the find method with an array of primary keys, which will return a collection of the matching records:

$flights = App\Flight::find([1, 2, 3]);

Ezek működnek anélkül is, hogy variable-be beraknám:

$post=App\Post::where(„id”, „>”, 5)->get();

App\Post::where(„id”, „>”, 5)->get();

Csak ha variable, akkor pl echo-zhatom vagy variálhatok vele.

Legtöbbször praktikusabb egy bejegyzést visszakérni csak, mert azzal könnyebben tudok bánni:

$post=App\Post::where(„id”, 5)->first();

$post->name = „New name”;

## Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The findOrFail and firstOrFail methods will retrieve the first result of the query; however, if no result is found, a Illuminate\Database\Eloquent\ModelNotFoundException will be thrown:

$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();

If the exception is not caught, a 404 HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return 404 responses when using these methods:

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

return App\Flight::findOrFail($id);

});

## Retrieving Aggregates

You may also use the count, sum, max, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');

Hogy ezt ne csak a tinkerben élvezhessük, csináljunk egy route-ot

routes=>web.php

(nem kell újat csinálni, az ott lévőt írjuk át)

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

return redirect(action("PostController@index"));

});

És akkor csináljuk is meg a PostControllert

cmd:

php artisan make:controller --resource PostController

Ez a „--resource” csak egy plusz hasznosság, így az új controllerünk több alapmethoddal rendelkezik.

A REST standard írja elő, hogy hogyan kell a route-ba methodokat írni,ezek az alapmethodok, ha mi akarunk újabbakat írni, akkro ne így hívjuk őket.

Úgyhogy használjuk ezt:

Route::resource("posts", "PostController");

Ettől tudjuk az egyes posztokat majd megnézni.

Kész a PostControllerünk. Hogy meg tudjuk mutatni a blogposztokat, ezt kell beleírni

a tetejére: use App\Post;

utána:

public function index()

{

$posts = Post::all();

return $posts;

}

De ez persze gagyi, mert Jsonban adja vissza, nem ezt akarjuk, hanem normális látványt.

resources->views folderben csinálni egy új foldert: post névvel és azon belül egy index.blade.php file-t

Így már korrektebb:

public function index()

{

$posts = Post::all();

return view("post.index", ["posts" => $posts]);

}

post.index => a post folderben lévő indexet adja vissza.

Lehet így is írni: post/index

["posts" => $posts]=> így adom át view-nak, milyen adatokat akarok látni. A view ugyanis semmilyen kapcsolatban nincs a modellel, valahogy oda kell neki adni az adatokat.

A view-nak át lett adva az adat és ott így dolgozzuk fel:

<ul>

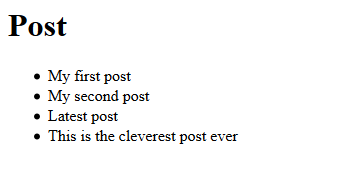
@foreach($posts as $post)

<li>{{$post->name}}</li>

@endforeach

</ul>

Ezt látom most az oldalon:



Tök jó, de kéne link, hogy ki tudjam nyitni a posztokat:

Hardkódolhatom a linkeket, de ezt ugye nem akarjuk, úgyhogy

{{action(„PostController@show”)}}

@foreach($posts as $post)

<li>

<a href="{{ action ("PostController@show", [$post->id]) }}">{{$post->name}}</a>

</li>

@endforeach

action ("PostController@show", [$post->id]) = ettől ő tudja, hogy a link vége postcontroller – show és melyik id-jű posztot adja vissza.

A PostControllerbe ezt írjuk be, hogy megszerezzük a szükséges id-t.

public function show($id)

{

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

return view("post.show", ["post" => $post]);

}

return view("post.show", ["post" => $post]);

Ezt többféleképpen írhatjuk, ugyanaz:

return view("post.show")->with(["post" => $post]);

return view("post.show", compact(["post"]));

Új poszt írása:

A post folderben egy create.blade.php file-t csinálok ezzel a tartalommal:

<form method="post" action"{{ action ("PostController@store") }}">

{{ csrf\_field() }}

<input type="text" name="name">

<textarea rows="10" name="content"></textarea>

<button type="submit">Create</button>

</form>

action"{{ action ("PostController@store") }}" => store , mert a PostControllerben így hívják alapból a módszert

A PostControllerben kibővítem a create functiont

public function create()

{

//

return view("post.create");

}

A formban ez a sor {{ csrf\_field() }} arra való, hogy egy tokent csináljon, hogy idegen oldalról ne lehessen beküldeni az adatokat az adatbázisomba.

A PostControllerben a store functiont is updatelem:

public function store(Request $request)

{

//

$name = $request -> name;

$content = $request -> content;

$post = new Post();

$post -> name = $name;

$post -> content = $content;

$post -> user\_id = $user\_id;

$post -> save();

return redirect(action("PostController@index"));

}

## Mass Assignment

You may also use the create method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a fillable or guarded attribute on the model, as all Eloquent models protect against mass-assignment by default.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the $fillable property on the model. For example, let's make the name attribute of our Flight model mass assignable:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* The attributes that are mass assignable.

\*

\* @var array

\*/

protected $fillable = ['name'];

}

Once we have made the attributes mass assignable, we can use the create method to insert a new record in the database. The create method returns the saved model instance:

$flight = App\Flight::create(['name' => 'Flight 10']);

If you already have a model instance, you may use the fill method to populate it with an array of attributes:

$flight->fill(['name' => 'Flight 22']);

Szóval, a post.php-ban (App->providers->post.php) megnevezzük, hogy mik a fillable mezők:

class Post extends Model

{

//

protected $fillable = ["name", "content", "user\_id"];

}

és ehelyett:

public function store(Request $request)

{

//

$name = $request -> name;

$content = $request -> content;

$post = new Post();

$post -> name = $name;

$post -> content = $content;

$post -> user\_id = $user\_id;

$post -> save();

return redirect(action("PostController@index"));

}

írhatjuk ezt:

public function store(Request $request)

{

$post = Post::create([

"name" => $name,

"content" => $content,

"user\_id" => $user\_id,

]);

return redirect(action("PostController@index"));

}

És hogy legyen is mit megmutatni, a views/post folderbe csinálunk egy show.blade.php file-t ezzel a tartalommal:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Blog</title>

</head>

<body>

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

<ul>

<p>{{$post->content}}</p>

</ul>

<a href="{{ action ("PostController@index")}}">back to index</a>

</body>

</html>

Adding Bootstrap to the project.

First download it.

<http://getbootstrap.com/getting-started/#download>

Nem kell unzippelni, három folder van benne, ezeket kimásolom, a public folderbe.

Ezek után

az index.blade.php-be

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

Ez az asset method az absolute urlt megcsinálta nekem.

A képeknél is ezt az asset methodot kell használni!

Authentication

cmd line-ba beírni:

php artisan make:auth

Ettől lesz egy új function a web.php-ban.

Ez:

Auth::routes();

És multiple controllers (forgot password, login etc) és views (passwords etc).

Összerakni oldalakat:

views alatt csinálok egy layout.blade.php –t

amiben sorban hívom az oldal részeit és itt van a HTML struktúra is:

<!doctype **html**>  
<**html lang="en"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**meta name="viewport"  
 content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0"**>  
 <**meta http-equiv="X-UA-Compatible" content="ie=edge"**>  
 <**title**>**@yield("title")**</**title**>  
</**head**>  
<**body**>  
  
**@include("header")**;

**@include("navigation")**;  
  
<**class class="container"**>  
 **@yield("main-content")**;  
</**class**>  
  
**@include("footer")**;  
  
<**script  
 src="http://code.jquery.com/jquery-3.1.1.min.js"  
 integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="  
 crossorigin="anonymous"**></**script**>  
  
</**body**>  
</**html**>

Hogy ez működjön, meg kell csinálni ezeket az oldalakat.

navigation.blade.php

stb

szintén a views folderben és ezek tartalmazzák a szükséges HTML kódot.

Ez a rész meg mindig behívja az aktuális oldal tartalmát.

pl navigation.blade.php:

<**a href="#"**>About</**a**>  
<**a href="#"**>Contact</**a**>

A @yield title és a main-content meg innen jön:

**@extends("layout")  
  
@section("title"**, **"My homepage")  
  
@section("main-content")**<**h1**>This is my blog</**h1**>  
  
<**ul**>  
 **@foreach(**$posts **as** $post**)** <**li**>  
 <**a href="**{{ action (**"PostController@show"**, [$post->**id**]) }}**"**>{{$post->**title**}}</**a**>  
 </**li**>  
 **@endforeach**</**ul**>  
  
**@endsection**

Minden oldal úgy kezdődik, hogy extends layout (kivéve a navigation meg a footer, mert az a layoutban már benne van és akkor kétszer hívná).

Az oldal tetején ezt is megadom:

**@section("title"**, **"My homepage")**

title azt jelenti, hogy a HTML melyik része, a második pedig, hogy az a rész mi legyen.

Így tudja a layout, hogy ezen az oldalon a title homepage

<**title**>**@yield("title")**</**title**>

És minden oldalon a lényeget becsomagolom „main-content” sectionbe:

**@section("main-content")**<**h1**>This is my blog</**h1**>  
  
<**ul**>  
 **@foreach(**$posts **as** $post**)** <**li**>  
 <**a href="**{{ action (**"PostController@show"**, [$post->**id**]) }}**"**>{{$post->**title**}}</**a**>  
 </**li**>  
 **@endforeach**</**ul**>  
  
**@endsection**

Innen tudja a layout, hogy ezt kell megmutatni:

<**class class="container"**>  
 **@yield("main-content")**;  
</**class**>

Ezn logika alapján a show.blade.php-t is ki lehet erre javítani:

**@extends("layout")  
  
@section("title"**, **"Individual posts")  
  
@section("main-content")**<**h1**>{{$post->**title**}}</**h1**>  
<**ul**>  
 <**p**>{{$post->**content**}}</**p**>  
</**ul**>  
  
<**a href="**{{ action (**"PostController@index"**)}}**"**>Back to main page</**a**>  
  
**@endsection**