Создаем новую таблицу в БД, используя миграции.

Синтаксис:

php artisan make:migration migration\_name --create="table\_name"

Нам надо создать таблицу litters с записями: id автоинкремент, integer, первичный ключ; litter – строковая, 2 символа .

php artisan make:migration create\_litters\_table --create="litters"

Дожидаемся подтверждения создания миграции.

Находим файл здесь\_будет\_дата\_и\_штамп\_времени \_create\_litters\_table.php миграции в папке database\migrations и начинаем его редактировать.

<?php

use Illuminate\Support\Facades\Schema;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateLittersTable extends Migration

{

public function up()

{

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

$table->increments('id');

$table->string('litter', 2);

$table->string('photo1');

$table->string('photo2');

$table->integer('idDog1', false);

$table->integer('idDog2', false);

$table->timestamps();

});

}

public function down()

{

Schema::dropIfExists('litters');

}

}

И запускаем нашу миграцию командой

php artisan migrate

Если при применении миграции вы получаете ошибку «class not found» («класс не найден»), попробуйте выполнить команду composer dump-autoload и заново запустить команду migrate.

|  |
| --- |
| **Принудительные миграции в продакшене****:**  Некоторые операции миграций разрушительны, это значит что они могут привести к потере ваших данных. Для предотвращения случайного запуска этих команд на вашей боевой БД перед их выполнением запрашивается подтверждение. Для принудительного запуска команд без подтверждения используйте ключ --force:  php artisan migrate --force  Как обычно для unix систем просмотреть подсказки по доступным опциям команды migrate можно вызвав справку:  php artisan migrate –h  Вот еще полезные опции данной команды:  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  [**Откат**](http://laravel.su/docs/5.0/migrations#rolling-back-migrations) **миграций:**  Отмена изменений последней миграции  php artisan migrate:rollback  Отмена изменений всех миграций  php artisan migrate:reset  Откат всех миграций и их повторное применение  php artisan migrate:refresh  php artisan migrate:refresh –seed |

Сразу создадим Модель для нашей таблицы:

php artisan make:model Litter

И отредактируем app\Litter.php

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Litter extends Model

{

protected $fillable = ["litter","photo1","photo2","idDog1","idDog2"];

protected $primaryKey = "id";

}

Мы можем для тестирования использовать механизм наполнения таблиц БД начальными данными с помощью специальных классов. Они создаются и хранятся в директории database\seeds.

Создадим класс для добавления данных в только что созданную таблицу нашей БД.

php artisan make:seeder LitterTableSeeder

Теперь открываем новый класс в database\seeds\LitterTableSeeder.php

И правим класс, занося наши данные:

<?php

use Illuminate\Database\Seeder;

use App\Litter;

class LitterTableSeeder extends Seeder

{

public function run()

{

DB::statement("SET foreign\_key\_checks=0");

Litter::truncate();

DB::statement("SET foreign\_key\_checks=1");

factory(Litter::class, 10)->create();

}

}

Теперь исправим фабрику данных, изменив файл database\factories\ModelFactory.php:

<?php

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

$faker = Faker\Factory::create('ru\_RU');

return [

'title' => $faker-> sentence,

'pageContent' => $faker-> realText . ' || ' . $faker-> realText,

'photo' => $faker->imageUrl($width = 960, $height = 427),

];

});

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

**$faker = Faker\Factory::create('ru\_RU');**

**return [**

**'litter' => $faker->randomElement($array =**

**array ("A","B","C","D","E","F","G")),**

**'photo1' => $faker->imageUrl($width = 450, $height = 600),**

**'photo2' => $faker->imageUrl($width = 450, $height = 600),**

**'idDog1' => $faker->numberBetween($min = 1, $max = 14),**

**'idDog2' => $faker->numberBetween($min = 1, $max = 14),**

**];**

**});**

Внесем изменения в уже имеющемся во фреймворке классе DatabaseSeeder, чтобы контролировать последовательное наполнение данными наши таблицы. Откроем для этого файл database\seeds\DatabaseSeeder.php и исправим его:

<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder

{

public function run()

{

$this->call('ContentTableSeeder');

$this->command->info('operation completed successfully!');

**$this->call('LitterTableSeeder');**

**$this->command->info(‘operation completed successfully!’);**

}

}

Сохраним его и запустим в командной строке:

php artisan db:seed

Это действие вызовет на исполнение класс DatabaseSeeder, в котором произойдет вызов класса LittersTableSeeder, произойдет наполнение данными, и выйдет сообщение о заполнении соответствующей таблицы. Если мы будем наполнять несколько таблиц, то необходимо создать отдельные классы для каждого действия по наполнению соответствующей таблицы, а потом указать вызовы этих классов в DatabaseSeeder в той последовательности, в которой мы хотим производить их наполнение.

Теперь нам надо связать между собой таблицы: litters с our\_dogs. Сделаем это и с помощью SQLite и в моделях app\Litter.php и в app\Our\_dog.php.

app\Litter.php:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Support\Facades\DB;

class Litter extends Model

{

protected $fillable = ["litter","photo1","photo2","idDog1","idDog2"];

protected $primaryKey = "id";

public static function mainPage() {

return DB::table("litters")->get();

}

}

Добавляем маршрут в routes\web.php:

Route::get('/litters', 'LitterController@index');

И создадим новый контроллер командой в Консоли:

php artisan make:controller LitterController

Необходимо отредактировать файл app\Http\Controllers\LitterController.php:

<?php

namespace App\Http\Controllers;

use App\Litter;

class LitterController extends Controller

{

public function index() {

return view("litters", ["litt" => Litter::mainPage()]);

}

}

Создаем представление resources\views\litters.balde.php и приведем его к виду:

@extends('layouts.app')

@section("title", "Помёты")

@section("main")

<p>

@if (auth()->check())

<a href="{{ route('litter.create') }}" class="btn btn-outline-info">Создать новый помёт</a></p>

@endif

<?php

function rndColor() {

$arrColor = ['#F2B4F2','#D4D4FB','#B4FCBA','#B4F6FC','#FCB4F6','#FCBAB4'];

$i = (mt\_rand(1, count($arrColor)) - 1);

echo $arrColor[$i];

}

?>

<div class="container">

@foreach ($litt as $litters)

<div class="card card-body" style="background-color:<?= rndColor(); ?>;">

<div class="row align-items-center text-center">

<div class="col-2">

<img class="h-n" src="{{ $litters->photo1 }}">

</div>

<div class="col-3 text-left">

{{ $litters->idDog1 }}

</div>

<div class="col-2">

{{ $litters->litter }}

</div>

<div class="col-3 text-right">

{{ $litters->idDog2 }}

</div>

<div class="col-2">

<img class="h-n" src="{{ $litters->photo2 }}">

</div>

</div>

</div>

@endforeach

</div>

@endsection

Для тестирования пути к изображениям указаны без использования конструкции <?php public\_path();?>/

При внесении локальных изображений, такую конструкцию необходимо будет добавить. Можно конечно проверять начальные символы в строке они будут либо img, либо нет. И в зависимости от выполнения данного условия добавлять или не добавлять данную конструкцию.

Если вы авторизованный пользователь, то страница открываться не будет, так как мы еще не создали маршрут для добавления нового раздела, который указан у нас в строке 6. Для этого добавим в маршрут в rotes\web.php:

Route::get("/litter/create", "LitterController@input")->name("litter.create");

В данный момент у нас выбираются из таблицы БД значения id\_dogs, но нам же нужны не они, а то, что соответствует этим id. Мы можем сделать второй запрос в другую связанную таблицу и, имея два результата запроса, прокрутить в отдельных циклах, и сопоставить значения. Но давайте сделаем выборку из двух таблиц в одном запросе.

Изменим в моделе app\Litter.php:

public static function mainPage() {

**$res = DB::table('litters')**

**->select('our\_dogs.name AS name1', 'p.name AS name2', 'litters.\*')**

**->join('our\_dogs', 'our\_dogs.id\_dogs', 'litters.idDog1')**

**->join('our\_dogs as p', 'p.id\_dogs', 'litters.idDog2')**

**->get();**

**return $res;**

}

Сделаем вначале форму ввода и редактирования разделов этой страницы проекта.

Редактируем контроллер app\Http\Controllers\LitterController.php:

<?php

namespace App\Http\Controllers;

use App\Litter;

**use App\Our\_dog;**

class LitterController extends Controller

{

public function index() {

return view("litters", ["litt" => Litter::mainPage()]);

}

**public function input(Litter $litter) {**

**return view ("litters.input", ["litt" => $litter, "dog" => Our\_dog::all()]);**

**}**

}

А именно, добавляем метод input(). И заметьте, я добавил модель app\Our\_dog.php – это необходимо сделать, так как для формы ввода и редактирования запрашиваем дополнительные данные из другой таблицы ‘our\_dogs’. Их будем использовать при построении раскрывающихся списков.

Далее выполним в консоле команду, чтобы создать сервис-класс запроса:

php artisan make:request LitterRequest

Отредактируем данные под нашу будущую форму ввода/редактирования app\Http\Requests\LitterRequest.php:

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class LitterRequest extends FormRequest

{

public function authorize()

{

return false;

}

public function rules()

{

return [

'litter' => 'required|max:2|regex:/[a-z][A-Z]/',

'idDog1' => 'required',

'idDog2' => 'required',

];

}

public function messages() {

return [

'litter.required' => 'Это обязательное поле',

'litter.max' => 'Слишком много символов для этого поля',

'litter.regex' => 'Только латинские символы',

'idDog1.required' => 'Это обязательное поле',

'idDog2.required' => 'Это обязательное поле',

];

}

}

Создадим файл resources\views\litters\input.blade.php:

@extends('layouts.app')

<?php $pageTitle = ($litt->id) ? "Редактирование " . $content->id : "Добавление" ?>

@section("title", $pageTitle . " -- раздела")

@section("main")

<div class="container top60">

<div class="row">

<div class="col-11">

<h1>@if ($litt->id) Редактирование раздела {{ $litt->litter }}

@else Добавление раздела

@endif

</h1>

<form action="{{ action('LitterController@save') }}" method="POST" enctype="multipart/form-data">

@if ($litt->id)

{{ method\_field('PUT') }}

<input type="hidden" name="id" value="{{ old('id', $litt->id) }}">

@endif

{{ csrf\_field() }}

<div class="form-group">

<label for="litter" class="col-md-4 control-label">Название помёта:</label>

<input id="litter" type="text" placeholder="Название помёта" class="form-control" name="litter" value="{{ old('litter', $litt->title) }}" required>

</div>

<div class="form-group">

<label for="idDog1" class="col-md-4 control-label">Собака 1:</label>

<select class="form-control" id="idDog1" name="iDog1">

@foreach ($dog as $dogs)

<option value="{{ $dogs->id\_dogs }}">{{ $dogs->name }}</option>

@endforeach

</select>

<label for="photo1" class="col-md-4 control-label">Изображение собаки 1:</label>

<input id="photo1" type="file" class="form-control" name="photo1">

</div>

<div class="form-group">

<label for="idDog2" class="col-md-4 control-label">Собака 2:</label>

<select class="form-control" id="idDog2" name="iDog2">

@foreach ($dog as $dogs)

<option value="{{ $dogs->id\_dogs }}">{{ $dogs->name }}</option>

@endforeach

</select>

<label for="photo2" class="col-md-4 control-label">Изображение собаки 2:</label>

<input id="photo2" type="file" class="form-control" name="photo2">

</div>

<button type="submit" class="btn btn-primary">Сохранить</button>

<a href="#" class="btn btn-secondary" onClick="window.location.href=window.location.href">Отмена</a>

</form>

</div>

</div>

</div>

@endsection

Добавим в контроллер LitterController.php следующий метод:

public function save(LitterRequest $request) {

$time\_rep = time();

$exch1 = $request->file('photo1')->getClientOriginalExtension();

$exch2 = $request->file('photo2')->getClientOriginalExtension();

$name1 = $time\_rep . '\_mary-dance\_ru' . '.' . $exch1;

$name2 = $time\_rep . '\_mary-dance\_ru' . '.' . $exch2;

$path1 = $request->file('photo1')->storeAs('img/litter', $name1, 'my\_files');

$path2 = $request->file('photo2')->storeAs('img/litter', $name2, 'my\_files');

if ($request->has("id")) {

$lit = DB::table('litters')->where('id', $request->id)->update([

'litter' => $request->litter,

'photo1' => $path1,

'photo2' => $path2,

'idDog1' => $request->idDog1,

'idDog2' => $request->idDog2,

]);

$s = " изменен";

} else {

$lit = DB::table('litters')->insert([

[

'litter' => $request->litter,

'photo1' => $path1,

'photo2' => $path2,

'idDog1' => $request->idDog1,

'idDog2' => $request->idDog2,

]

]);

$s = " добавлен";

}

return redirect()->action("LitterController@index")

->with("status", "Запись " . $request->litter . $s);

}

И обязательно добавим маршруты для методов POST и PUT:

Route::match(["post", "put"], "/cont", "MainController@save");

Создадим представление, в котором будем открывать каждый раздел страницы litter.blade.php. Постараемся вместить туда описание и фотогалерею.

Создадим каталог resources\letters и файл view.blade.php в этом каталоге.