Обязательно добавить везде фиксации в Git после каждого серьезного именения.

Разработка через тестирование.

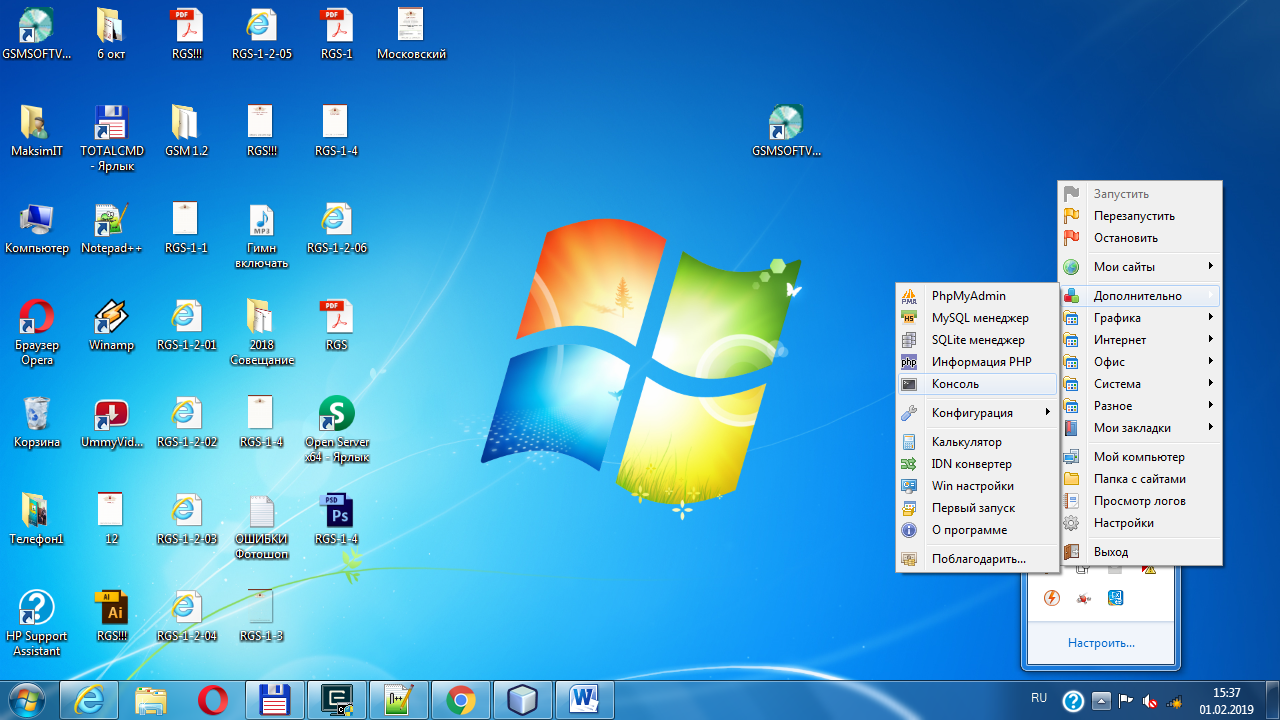
В этой части сделаем условно-динамический контент главной страницы. Мы создадим возможность изменения текста разделов на главной странице.

Определимся вначале со структурой. Создадим Карусель, которая будет растянута по всей ширине окна (шириной Карусели, ее автоматическим изменением размера под разные типы экранов и прочими ее фишками за нас будет делать используемый нами CSS-фреймворк – Bootstrap 4). Карусель будет состоять из 3-х слайдов, под которые мы выберем три фотографии. Каждый слайд будет как бы презентацией трех информационных разделов нашего сайта, и сами разделы поместим ниже Карусели с полным текстом. В Карусели же у нас будет усеченный текст каждого раздела. И не забудем добавить в представление созданные ранее и автоматически подставляемые «слои» панели навигации и футера.

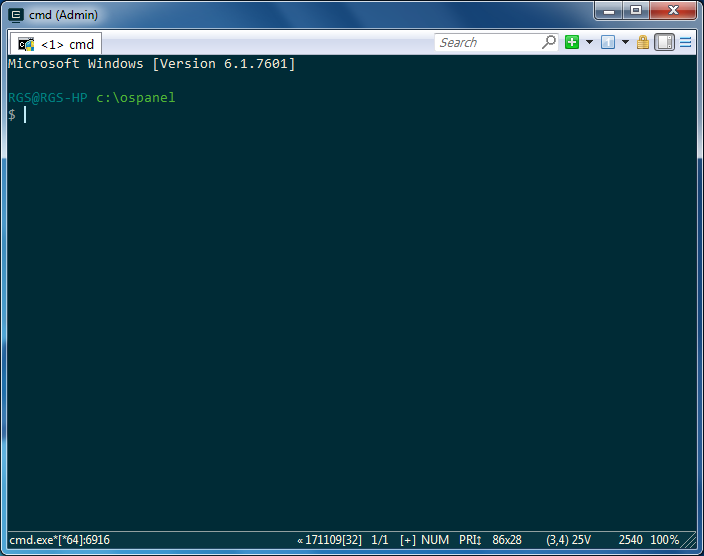
Создадим сразу таблицу базы данных content с помощью миграции. В таблице будет поле id – integer, unsigned, Автоматическое приращение. Что и станет первичным ключом. Второе поле: title – здесь будет название раздела и соответственно слайда – varchar (строковое значение) можем определить ему размер символов, если у нас есть представление о том, какой длинны названия будет создавать администратор сайта, но мы оставим значение по-умолчанию. Третьим полем siteContent станет основное содержание раздела, которое может быть большим по размеру, поэтому выбирая из TEXT и MEDIUMTEXT (см. памятку по размерам строковых полей) я выбрал TEXT, т.к. слишком много текста в разделах будет довольно утомительно для посетителей. Ну и не будем отказываться от предлагаемых нам полей создания записи и ее модификации, на размер нашей небольшой таблицы это сильно не повлияет, а нам может пригодиться.

|  |  |
| --- | --- |
| **Памятка по размерам строковых полей** | |
| VARCHAR | Может хранить не более 255 символов. |
| TINYTEXT | Может хранить не более 255 символов. |
| TEXT | Может хранить не более 65 535 символов. |
| MEDIUMTEXT | Может хранить не более 16 777 215 символов. |
| LONGTEXT | Может хранить не более 4 294 967 295 символов. |

Чтобы открыть консоль нам надо развернуть панель Инструменты OpenServer’а в трее и выбирать Дополнительно – Консоль



Открывается следующее окно:



Мы находимся в корне папки OpenServer, в данном случае в OSPanel – название папки зависит от версии пакета OpenServer и его установки.

Переходим в наш проект командой:

cd domains\mary-dance.lc

Создаем модель командой

php artisan make:model Content -m

Эта команда создаст одновременно модель и миграцию

Правим код в файле create\_conents\_table.php

<?php

use Illuminate\Support\Facades\Schema;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateContentsTable extends Migration

{

public function up()

{

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

$table->increments('id');

$table->string('title');

$table->text('pageContent');

$table->timestamps();

});

}

public function down()

{

Schema::dropIfExists('contents');

}

}

И запускаем миграцию, а модель пока не трогаем.

php artisan migrate

Таблица создана, теперь наполним ее первичными данными, но не вручную, как мы это делали ранее, а с помощью библиотеки faker.

Создадим фабрику моделей для автоматического наполнения первичными данными нашей базы данных. Мы будем использовать в очень популярную библиотеку faker, которая генерирует разнообразные фейковые (следуя из названия) данные. Данную библиотеку часто используют для модульного и unit-тестирования при разработке. По-умолчанию используется английский язык и форматы данных в выводе, однако существует возможность переопределить данные на другие языки вывода.

Разделы с типами данных, генерируемых библиотекой.

* **Base:** простые методы для случайных букв, чисел, обработанных строк и регулярных выражений
* **Lorem Ipsum Text:** случайный текст на латыни
* **Person:** имена людей
* **Address:** почтовые адреса
* **Phone Number:** номера телефонов
* **Company:** имена компаний
* **Real Text:** фактический текст, написанный людьми вместо бессмысленных латинских строк
* **Date and Time:** случайные даты и время
* **Internet:** электронная почта, домены и т.д.
* **User Agent:** строки браузера
* **Payment:** строки и номера кредитных карт и SWIFT
* **Color:** случайные цвета
* **File:** расширения файлов, типы файлов и имена файлов
* **Image:** URL-адреса изображений-заполнителей разных видов
* **Uuid:** уникальные IDs
* **Barcode:** различные типы штрих-кодов, например. ISBN13
* **Miscellaneous:** коды шифрования, коды стран и т.д.
* **Biased:** случайные числа с уклоном

Фабрика может генерировать данные без записи в БД – метод make() (что используется для тестирования), а может, чем мы и воспользуемся, сохранить эти фейковые данные в базу данных – метод create().

Для этого те, кто используют версию Laravel от 5.5 и далее могут выполнить команду

php artisan make:factory Content

и отредактировать созданный файл database\factories\Content.php и запускать каждую фабрику по-отдельности, но т.к. мы работаем с версией 5.4, то мы все фабрики создаем в имеющемся файле database\factories\ ModelFactory.php. Откроем его и добавим к уже имеющемуся коду следующий фрагмент:

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

return [

'title' => $faker-> sentence,

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

];

});

В строке *'pageContent' => $faker-> realText . ' || ' . $faker-> realText,* я увеличиваю количество данных простой конкатенацией (соединением текста). Можно конечно использовать paragraphs, но он выдаст случайный текст на латыни, а мне хочется почитать что-то что я смогу прочесть. Т.е. вы можете использовать любой свой вариант, а я предпочел этот.

Создадим seeder класс для наполнения данными

php artisan make:seeder ContentTableSeeder

Отредактируем файл database\seeds\ContentTableSeeder.php

<?php

use Illuminate\Database\Seeder;

use Illuminate\Support\Facades\DB;

use App\Content;

class ContentTableSeeder extends Seeder

{

public function run()

{

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

Content::truncate();

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

factory(Content::class, 3)->create();

}

}

Снимаем защиту первичного ключа

SET foreign\_key\_checks=0

Очищаем таблицу если в ней уже были какие-то данные, используя нашу модель Content:

Content::truncate();

Устанавливаем защиту обратно

SET foreign\_key\_checks=1

И запускаем фабрику моделей на обращение к модели три раза - цифра 3 указанная вторым параметром, что создаст три записи в БД. Метод create() в отличие от метода make() не только заполнит данными результат запроса, но и внесет эти данные в нашу таблицу.

<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder

{

public function run()

{

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

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

$this->call('ContentTableSeeder');

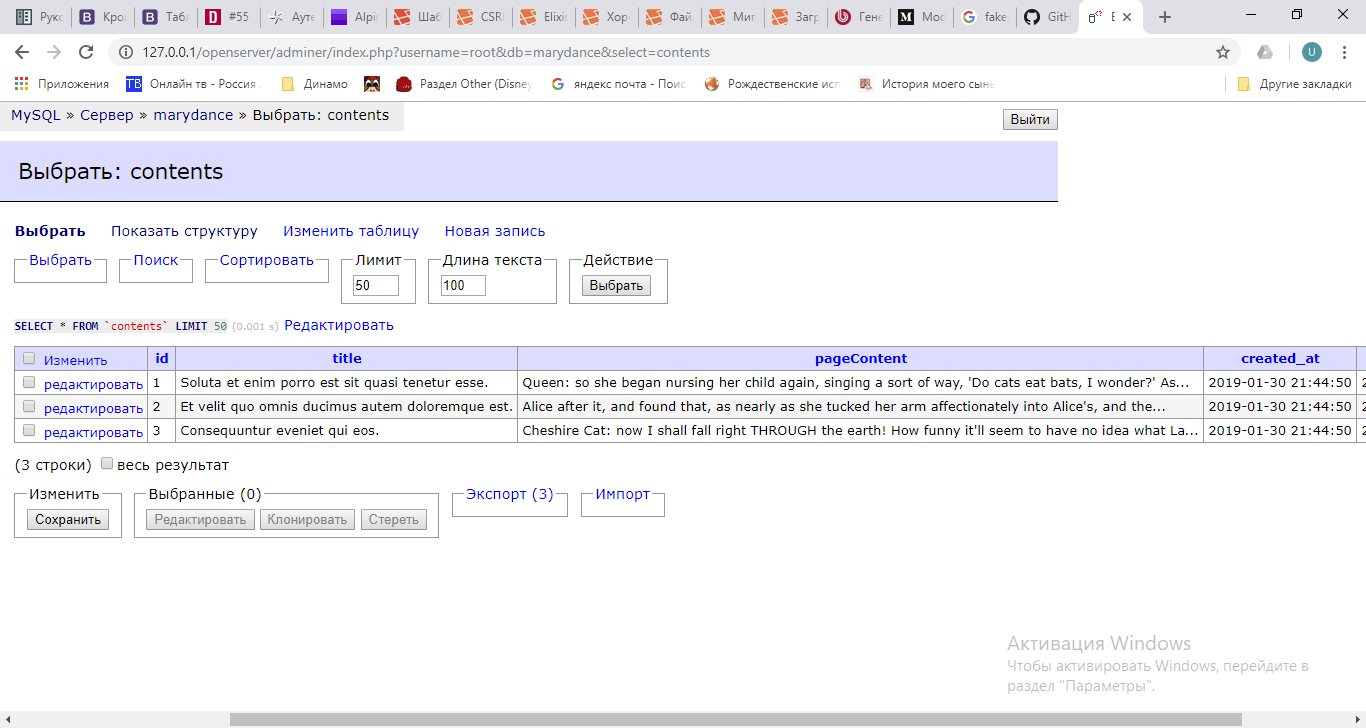
$this->command->info('operation completed successfully!');

}

}

Сохраним его и запустим в командной строке:

php artisan db:seed



На картинке показан результат всех наших действий.

Таблица создана, модель готова, данные заполнены, - двигаемся дальше и переходим к созданию маршрута, контроллера и представления.

Открываем файл с маршрутами routes\web.php

Если у нас уже есть действие для маршрута ‘/’, например, Route::get('/', 'DogController@first');

то редактируем его, если нет, то создаем новый маршрут:

Route::get('/', 'MainController@index');

Сохраняем и создаем контроллер командой

php artisan make:controller MainController

Если же у нас такой контроллер уже существует, то правим его код (котроллер у нас находится в файле app\Http\Controllers\MainController.php):

<?php

namespace App\Http\Controllers;

class MainController extends Controller

{

public function input() {

return view("main");

}

}

Такое простое действие контроллера будет вызывать файл представления, расположенный в resources\views\main.blade.php

В название файла часть blade показывает, что мы используем Blade-шаблоны для создания представлений.

Приведу пример изначального кода представления:

@extends('layouts.app')

@section("title", "Главная")

@section("main")

<div id="carouselMainControls" class="carousel slide carousel-fade" data-ride="carousel">

<ol class="carousel-indicators">

<li data-target="#carouselExampleIndicators" data-slide-to="0" class="active"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="1"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="2"></li>

</ol>

<div class="carousel-inner">

<div class="carousel-item active">

<img class="d-block w-100" src="carousel/1.jpg" alt="First slide">

<div class="carousel-caption d-none d-md-block">

<h5>ПЕРВЫЙ СЛАЙД</h5>

<p>Всякое описание раздела</p>

</div>

</div>

<div class="carousel-item">

<img class="d-block w-100" src="carousel/2.jpg" alt="First slide">

<div class="carousel-caption d-none d-md-block">

<h5>ВТОРОЙ СЛАЙД</h5>

<p>Всякое описание раздела</p>

</div>

</div>

<div class="carousel-item">

<img class="d-block w-100" src="carousel/3.jpg" alt="First slide">

<div class="carousel-caption d-none d-md-block">

<h5>ТРЕТИЙ СЛАЙД</h5>

<p>Всякое описание раздела</p>

</div>

</div>

</div>

<a class="carousel-control-prev" href="#carouselMainControls" role="button" data-slide="prev">

<span class="carousel-control-prev-icon" aria-hidden="true"></span>

<span class="sr-only">Previous</span>

</a>

<a class="carousel-control-next" href="#carouselMainControls" role="button" data-slide="next">

<span class="carousel-control-next-icon" aria-hidden="true"></span>

<span class="sr-only">Next</span>

</a>

</div>

<div class="container marketing">

<div class="row">

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 1</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста

Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста

Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста</p>

</div>

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 2</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста.</p>

</div>

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 3</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста.</p>

</div>

</div>

</div>

@endsection

Вкратце поясню. У нас есть слой app.blade.php, который указан в самом верху нашего представления. Там обозначено место, в котором размещается секция main. В этом же файле мы определяем секцию main и размещаем в ней код страницы. В первой части страницы, с помощью CSS-фреймворка Bootstrap4 мы создаем Карусель с картинками и текстом, во второй же части создаем разделы на странице предваренными серыми кружками, расположенными на большом экране в ряд, а на маленьких друг под другом. Теперь надо вместо статического текста создать вставки из таблицы contents.

Мы можем как и в документации использовать только контроллер и создать в только что созданном методе index запрос в БД и вернуть результат в вызов представления:

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

class MainController extends Controller

{

public function index() {

$cont = DB::table('contents')->select('title', 'pageContent')->get();

return view("main", ["content" => $cont]);

}

}

Используя метод table() фасада DB мы создаем SQL запрос, который с помощью метода get() вернет нам в переменную $cont объект полученный запросом из таблицы contents. Добавив метод select() мы ограничиваем запрос только на получение данных из title и pageContent. Затем передаем полученные данные в представление, где мы их сможем вызвать, обратившись к переменной $content в представлении main.blade.php, которой мы передали данные из $cont этой строчкой “content” => $cont

Вот код измененного представления main.blade.php

@extends('layouts.app')

@section("title", "Главная")

@section("main")

<div id="carouselMainControls" class="carousel slide carousel-fade" data-ride="carousel">

<ol class="carousel-indicators">

<li data-target="#carouselExampleIndicators" data-slide-to="0" class="active"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="1"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="2"></li>

</ol>

<div class="carousel-inner">

<?php $i=1; ?>

@foreach ($content as $cont)

<div class="carousel-item <?php if ($i==1) { echo 'active'; }?>">

<img class="d-block w-100" src="carousel/<?=$i;?>.jpg" alt="<?=$i;?> слайд">

<div class="carousel-caption d-none d-md-block">

<h5>{{ $cont->title }}</h5>

<p>

<?php echo (html\_entity\_decode(mb\_strimwidth($cont->pageContent, 0,200,"...",mb\_internal\_encoding()))); ?>

</p>

</div>

</div>

<?php $i++; ?>

@endforeach

</div>

<a class="carousel-control-prev" href="#carouselMainControls" role="button" data-slide="prev">

<span class="carousel-control-prev-icon" aria-hidden="true"></span>

<span class="sr-only">Previous</span>

</a>

<a class="carousel-control-next" href="#carouselMainControls" role="button" data-slide="next">

<span class="carousel-control-next-icon" aria-hidden="true"></span>

<span class="sr-only">Next</span>

</a>

</div>

<div class="container marketing">

<div class="row">

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 1</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста

Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста

Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста</p>

</div>

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 2</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста.</p>

</div>

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>Раздел 3</h2>

<p>Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста

Очень много текста Очень много текста Очень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текстаОчень много текста.</p>

</div>

</div>

</div>

@endsection

Мы внесли изменение в первую часть шаблона и некоторые моменты надо объяснить более подробно. Во-первых, создаем цикл для прохождения по всему нашему объекту $content, который был передан из модели с результатами запроса в БД. Мы используем синтаксис шаблонов Blade: @foreach ($content as $cont) что соответствуетзаписи на PHP <?php foreach ($content as $cont) { ?>

Т.к. изображениям у меня изначально были присвоены имена 1.jpg, 2.jpg, 3.jpg, то создаем переменную $i и присваиваем ей значение 1. И далее внутри цикла foreach делаем вызов изображения исходя из изменяющихся значений $i следующим образом:

<img class="d-block w-100" src="carousel/<?=$i;?>.jpg">

Отсюда имеем, что сколько строк из таблицы нам «придет» с запросом, столько прохождений цикла и будет сделано, соответственно столько изображений будет вставлено, если мы не забудем переименовать изображения соответствующим образом. Далее, когда мы начнем создавать форму создания и редактирования разделов главной страницы, мы добавим возможность загрузки новых изображений и сохранение их в нашей БД.

Вызов заголовка раздела производим с помощью синтаксиса Blade вывода на экран {{ $cont->title }}Такой синтаксис подразумевает предварительное экранирование данных функциями php. Для вывода усеченной части основного содержания каждого раздела используем следующую конструкцию php, в которой предварительно декодируем все HTML-сущности функцией html\_entity\_decode, т.к. в этих полях БД содержатся HTML-теги для форматирования:

<?php echo (html\_entity\_decode(mb\_strimwidth($cont->pageContent, 0, 200, "...", mb\_internal\_encoding()))); ?>

# Описание функции: mb\_strimwidth

(PHP 4 >= 4.0.6, PHP 5, PHP 7)

mb\_strimwidth — Получение строки, обрезанной до заданного размера

**Описание**

mb\_strimwidth ( string $str , int $start , int $width [, string $trimmarker = "" [, string$encoding = mb\_internal\_encoding() ]] ) : string

Обрезает строку ([string](http://php.net/manual/ru/language.types.string.php)) str до длины width символов.

**Список параметров**

**str**

Исходная строка.

**start**

Смещение от начала строки. Количество символов от начала строки (первый символ стоит в позиции 0). Если указано отрицательное число, то отсчет будет идти с конца строки.

**width**

Размер вырезаемой части в символах. Отрицательные значения отсчитываются с конца.

**trimmarker**

Строка, которая заместит конец обрезанной строки.

**encoding**

Параметр encoding представляет собой символьную кодировку. Если он опущен, вместо него будет использовано значение внутренней кодировки.

[**mb\_internal\_encoding()**](http://php.net/manual/ru/function.mb-internal-encoding.php)- Установка/получение внутренней кодировки скрипта

Используем функцию для работы с многобайтными символами для корректной обработки кириллицы. Задаем, что из $cont раздела pageContent выводим текст только начиная с позиции 0 до позиции 200, а после добавляется многоточие (…). Данная функция не просто обрезает текст на позиции 200, а обрезает текст возможно даже меньше 200 символа, чтобы последнее слово отобразилось целиком. Это более приятно для восприятия, чем иметь в выводе обрезанный те… (это яркий пример того, что бы было, пойдя мы другим путем).

А теперь поучимся проводить рефакторинг собственного кода. Это необязательно делать с таким небольшим sql-запросом в разрезе получаемых данных, т.е. мы могли бы его оставить и в контроллере, но я рекомендую научиться выносить лишний код из контроллеров в различные сервисные классы, а в нашем случае, в модель Content.php

Википедия:

**Рефа́кторинг** (англ. *refactoring*), или **перепроектирование кода, переработка кода, равносильное преобразование алгоритмов** — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Отредактируем ранее созданную модель app\Content.php и приведем ее к такому виду:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Support\Facades\DB;

class Content extends Model

{

public static function mainPage() {

return DB::table('contents')->select('title', 'pageContent')->get();

}

}

Контроллер MainController.php приведем к такому виду:

<?php

namespace App\Http\Controllers;

use App\Content;

class MainController extends Controller

{

public function index() {

return view("main", ["content" => Content::mainPage()]);

}

}

Вот так мы сделали наш контроллер «тоньше». Теперь мы из контроллера вызываем статический метод нашей модели, в котором формируется запрос и получается ответ из БД.

Вообще же следование SOLID архитектуре, концепции MVC – все это является благодатной почвой для большого количества споров по поводу: что не надо держать в контроллерах, куда и в какие именно сервисные классы, запросы и т.д. надо выносить код. Иногда большое распыление логики по разным классам и файлам приводит к еще большей путанице при поддержке и доработке такого проекта. Совет один: следуйте своему мнению и не перегибайте палку.

Давайте доработаем вторую часть нашего представления, а потом займемся таким вопросом: а не предоставить ли администратору сайта возможность изменять изображения, используемые в Карусели, и что нам для этого надо.

@extends('layouts.app')

@section("title", "Главная")

@section("main")

<div id="carouselMainControls" class="carousel slide carousel-fade" data-ride="carousel">

<ol class="carousel-indicators">

<li data-target="#carouselExampleIndicators" data-slide-to="0" class="active"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="1"></li>

<li data-target="#carouselExampleIndicators" data-slide-to="2"></li>

</ol>

<div class="carousel-inner">

<?php $i=1; ?>

@foreach ($content as $cont)

<div class="carousel-item <?php if ($i==1) { echo 'active'; }?>">

<img class="d-block w-100" src="carousel/<?=$i;?>.jpg" alt="<?=$i;?> слайд">

<div class="carousel-caption d-none d-md-block">

<h5>{{ $cont->title }}</h5>

<p>

<?php echo (html\_entity\_decode(mb\_strimwidth($cont->pageContent, 0, 200, "...", mb\_internal\_encoding()))); ?>

</p>

</div>

</div>

<?php $i++; ?>

@endforeach

</div>

<a class="carousel-control-prev" href="#carouselMainControls" role="button" data-slide="prev">

<span class="carousel-control-prev-icon" aria-hidden="true"></span>

<span class="sr-only">Previous</span>

</a>

<a class="carousel-control-next" href="#carouselMainControls" role="button" data-slide="next">

<span class="carousel-control-next-icon" aria-hidden="true"></span>

<span class="sr-only">Next</span>

</a>

</div>

<div class="container marketing">

<div class="row">

@foreach ($content as $cont)

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>{{ $cont->title }}</h2>

<p>{!! html\_entity\_decode($cont->pageContent) !!}</p>

</div>

@endforeach

</div>

</div>

@endsection

Код стал намного меньше и более удобочитаемый.

Запускаем нашу главную страницу, и любуемся проделанной работой.

Теперь нам надо создать страницу просмотра разделов в табличном виде (ее мы сделаем доступной только администраторам) с инструментами для добавления, редактирования и удаления разделов главной страницы. Потом сделаем форму для ввода и редактирования данных в разделах. Здесь же задумаемся над несколькими проблемами:

1. Давайте создадим возможность для администратора сайта менять изображения в нашей Карусели.
2. Не будем ограничивать главную страницу только тремя разделами, а сделаем возможность создавать большее количество разделов и изображений в Карусели.

Во-первых, добавим маршрут для отображения нашей будущей страницы. Для этого открываем маршруты и добавляем следующую строку:

Route::get("/content", "MainController@all");

Что означает, получив запрос вида http://наш\_сайт/content , маршрутизатор вызовет метод all() нашего контроллера MainController.

Добавим в контроллер app\Http\Controllers\MainController.php метод all():

public function all() {

$cont = DB::table("contents")->get();

return view("contents.all", ["cont" => $cont]);

}

Здесь мы делаем SQL-запрос в таблицу contents нашей БД, вида SELECT \* FROM `contents`;. Мы можем уменьшить количество получаемых данных, указав, что нам нужны только id, title, pageContent и photo, а остальное нас не интересует, чтобы уменьшить объем задействованной памяти. И хотя в виду малого количества данных в нашей таблице, нас это не сильно-то и волнует, но давайте сразу и сделаем это:

public function all() {

**$content = DB::table("contents")->select("id", "title", "pageContent", "photo")->get();**

return view("contents.all", ["cont" => **$content**]);

}

И так, мы получаем значения из полей id, title, pageContent и photo (создаем запрос вида: SELECT `id`, `title`, `pageContent`, `photo` FROM `contents`;) и передаем их в представление contents\all.blade.php через переменную $cont. На данном этапе у нас еще нет поля photo в таблице БД contents, нет папки contents в наших представлениях, и нет файла представления all.blade.php. Все это мы создадим далее, а сначала сделаем рефакторинг нашего кода в контроллере.

Вынесем работу по извлечению записей из БД в модель app\Content.php так, как мы это уже делали ранее.

Отредактируем ранее созданную модель app\Content.php и приведем ее к такому виду:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Support\Facades\DB;

class Content extends Model

{

public static function mainPage() {

return DB::table('contents')->select('title', 'pageContent')->get();

}

**public static function forEditConent() {**

**return DB::table("contents")->select("id", "title", "pageContent", "photo")->get();**

**}**

}

Добавили новый метод forEditContent() в нашу модель.

Изменим метод all() в контроллере MainController() таким образом:

public function all() {

return view("contents.all", ["cont" => Content::forEditConent()]);

}

Теперь вся работа по созданию запроса и извлечению данных из БД у нас делает модель, а контроллер передает эти данные в представление.

Сразу предлагаю изменить нашу таблицу contents – добавим в нее поле для изображений, раз мы уже начали реализовывать механизм изменения изображений в Карусели. Здесь нам не надо ничего придумывать, мы просто в каждой записи будем хранить помимо тех данных, что у нас есть, еще и данные об изображении. Добавим к таблице поле photo типа varchar(150) и NOTNULL.

Откроем Консоль, перейдем в папку с проектом и сделаем откат последней миграции командой:

php artisan migrate:rollback

Откроем файл с нашей миграцией database\migrations\дата\_метка\_времени\_create\_contents\_table.php и отредактируем его. Итоговый файл должен выглядеть так:

<?php

use Illuminate\Support\Facades\Schema;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateContentsTable extends Migration

{

public function up()

{

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

$table->increments('id');

$table->string('title');

$table->text('pageContent');

**$table->string('photo', 150);**

$table->timestamps();

});

}

public function down()

{

Schema::dropIfExists('contents');

}

}

Добавленную строку я выделил жирным шрифтом. В ней задаем формат varchar у поля и вторым аргументом (после запятой) задаем длину поля -150. Сохраняем и запускаем миграцию, выполнив в Консоле команду:

php artisan migrate

Таблица создана повторно, теперь опять наполним ее данными с помощью фабрики моделей и библиотеки faker.

Редактируем database\factories\ModelFactory.php

<?php

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

return [

'title' => $faker-> sentence,

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

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

];

});

Редактируем database\seeds\LitterTableSeeder.php

<?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");

Litter::create(array('litter' => 'A'));

}

}

Добавили очистку таблицы до запуска наполнения данными, чтобы не было задвоения строк.

Редактируем database\seeds\DatabaseSeeder.php

<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder

{

public function run()

{

$this->call('LitterTableSeeder');

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

$this->call('ContentTableSeeder');

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

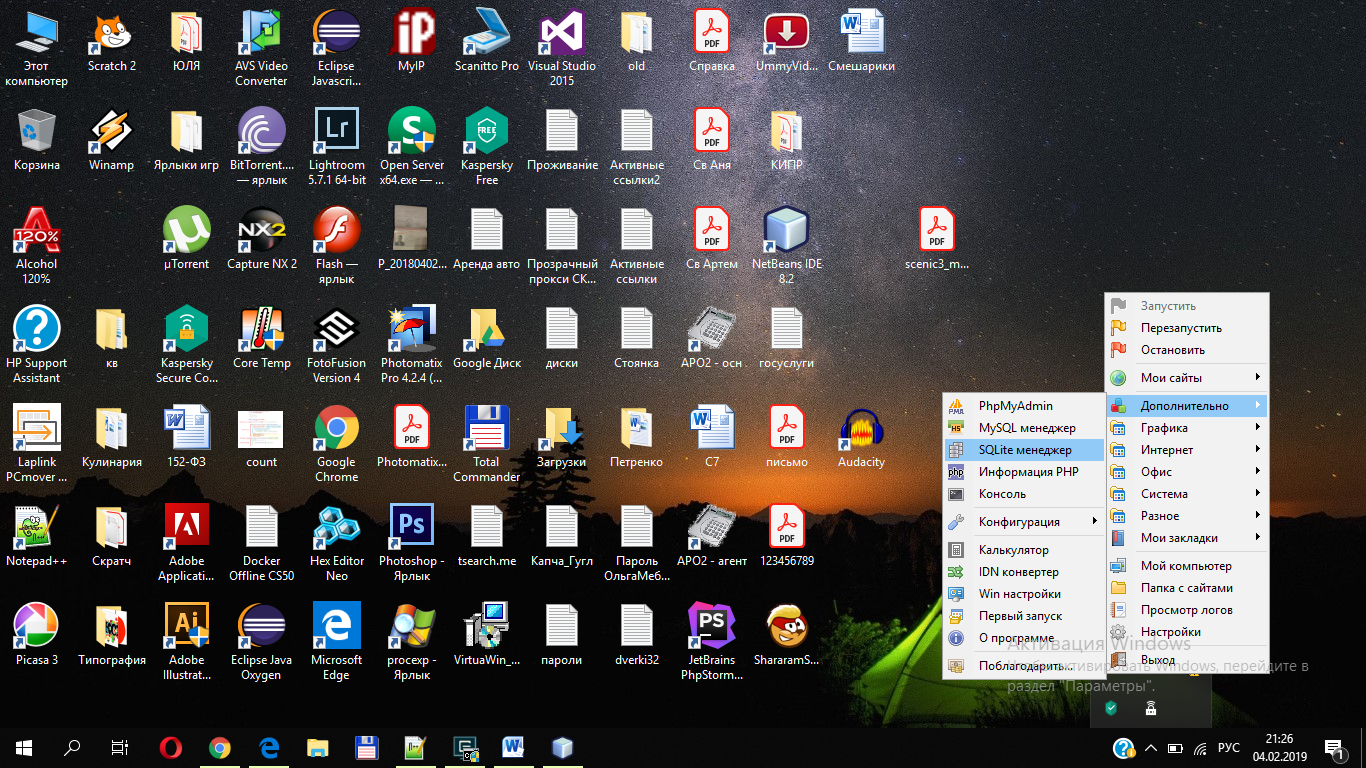
}

}

Запускаем в Консоле наполнение фейковыми данными:

php artisan db:seed

Все, наша таблица заполнена данными. Освоим еще один инструмент, который нам также может сильно упростить работу с БД, особенно в случае ее восстановления. Давайте откроем панель Инструменты OpenServer’а в трее и выберем Дополнительно – SQLite менеджер



Обязательно рассказать про первый запуск SQLite менеджер если про это не было сказано. И продолжить про Экспорт БД и дампа БД.

Добавив изображения в таблицу базы данных, нам необходимо вернуться к главной странице сайта и реализовать механизм отображения изображений в Карусели и формирование произвольного количества слайдов в Карусели, в зависимости от количества записей в таблице и от наличия в них записей с изображениями.

Создание sql-запроса, выборку и получение данных у нас делает модель app\Content.php, поэтому отредактируем в этой модели метод mainPage(). Необходимо добавить в sql-запрос поле photo. Заодно было бы неплохо знать количество записей в таблице. Для этого создадим в модели метод mainCount(), который с помощью выражения DB::table('contents')->count(); предоставить нам эти данные. После внесения изменений этот метод должен выглядеть следующим образом:

public static function mainPage() {

return DB::table('contents')->select('title', 'pageContent', 'photo')->get();

}

public static function mainCount() {

return DB::table('contents')->count();

}

Исправим метод index() в контроллере app\Http\Controllres\MainController.php:

public function index() {

return view("main", ["content" => Content::mainPage(), "count" => Content::mainCount()]);

}

Благодаря нашим изменениям, мы теперь передаем в представление resources\views\main.blade.php две переменные - $content, в которой содержится коллекция с результатом запроса в БД всех наших данных и $count, в которой содержится количество «строк» результата нашего запроса. Переменную $count мы добавили для прохождения цикла столько раз, сколько разделов у нас есть в таблице. Этим мы сформируем слайды Карусели.

Код нашей главной страницы resources\main.blade.php:

@extends('layouts.app')

@section("title", "Главная")

@section("main")

<div id="carouselMainControls" class="carousel slide carousel-fade" data-ride="carousel">

<ol class="carousel-indicators">

**@for($i = 0; $i < $count; $i++)**

**<li data-target="#carouselExampleIndicators" data-slide-to="$i" <?php if ($i==0) { echo 'class="active"'; }?>></li>**

**@endfor**

</ol>

<div class="carousel-inner">

**<?php $i=0; ?>**

@foreach ($content as $cont)

**<div class="carousel-item <?php if ($i==0) { echo 'active'; }?>">**

**<img class="d-block w-100" src="{{ $cont->photo }}">**

<div class="carousel-caption d-none d-md-block">

<h5>{{ $cont->title }}</h5>

<p>

<?php echo (html\_entity\_decode(mb\_strimwidth($cont->pageContent,0,200,"...",mb\_internal\_encoding()))); ?>

</p>

</div>

</div>

<?php $i++; ?>

@endforeach

</div>

<a class="carousel-control-prev" href="#carouselMainControls" role="button" data-slide="prev">

<span class="carousel-control-prev-icon" aria-hidden="true"></span>

<span class="sr-only">Previous</span>

</a>

<a class="carousel-control-next" href="#carouselMainControls" role="button" data-slide="next">

<span class="carousel-control-next-icon" aria-hidden="true"></span>

<span class="sr-only">Next</span>

</a>

</div>

<div class="container marketing">

<div class="row">

@foreach ($content as $cont)

<div class="col-lg-4">

<img class="rounded-circle" src="data:image/gif;base64,R0lGODlhAQABAIAAAHd3dwAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==" alt="Generic placeholder image" width="140" height="140">

<h2>{{ $cont->title }}</h2>

<p>{!! html\_entity\_decode($cont->pageContent) !!}</p>

</div>

@endforeach

</div>

</div>

@endsection

Внесенные изменения выделены.

Следующим этапом создадим папку contents в видах для отделения всех вспомогательных страниц от главных страниц проекта. Существует множество способов создания папок и документов в них, ну а мы пойдем по более профессиональному пути – через консоль, которую нам предоставляет OpenServer и которая является усредненным инструментом между cmd в Windows и консолью Linux-систем, т.к. поддерживает различный синтаксис.

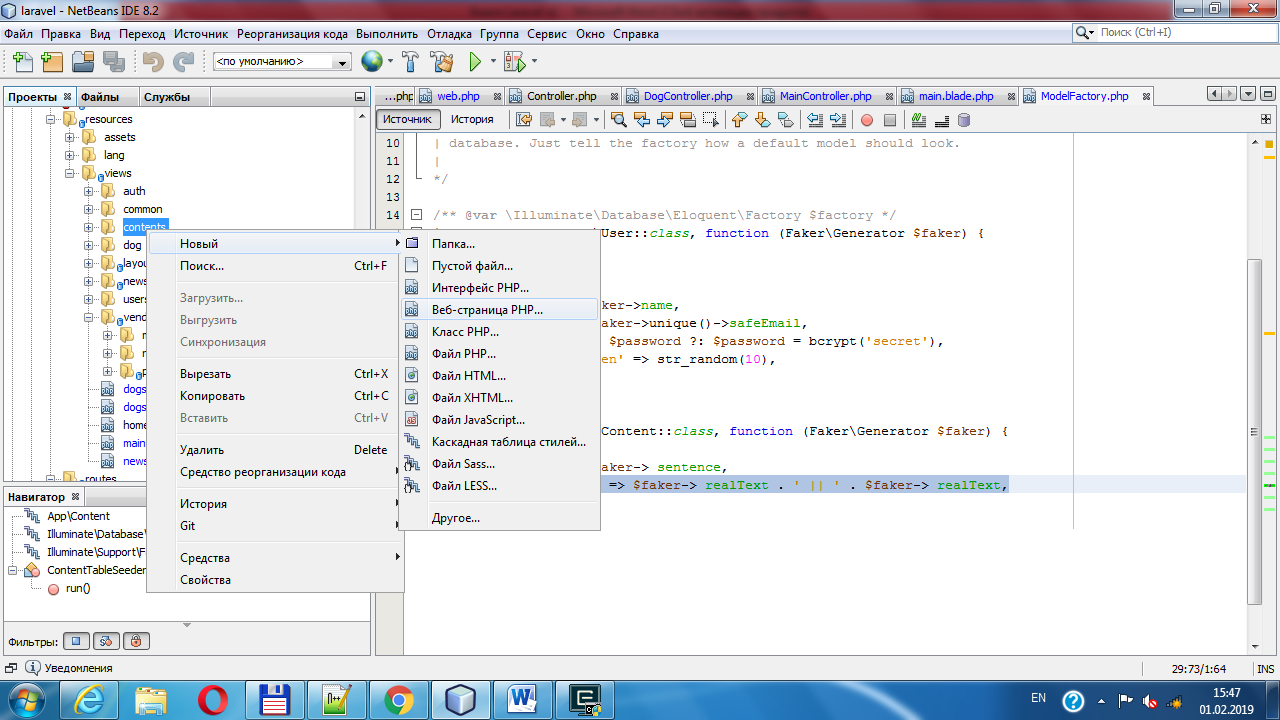
Запускаем Консоль OpenServser’a как мы это делали ранее, и если мы находимся в корне папки OpenServer, то переходим в папку с представлениями нашего проекта:

cd domains\mary-dance.lc\resources\views

И создаем новую папку contents:

mkdir contents

Чтобы создать страницу табличного вывода всех имеющихся на данный момент разделов главной страницы, перейдем в наш IDE NetBeans и щелкнув правой клавишей мыши по папке contents в видах, выберем Новый – Веб-страница PHP (или Пустой Файл)



В открывшейся форме нам надо заполнить только Имя файла (напишем all.blade.php) и проверить его расположение. Щелкнем Готово, и файл создан.

Теперь откроем созданную страницу resources\views\contents\all.blade.php для редактирования, удалим имеющийся там код и заполним страницу:

@extends('layouts.app')

@section("title", "Все разделы")

@section("main")

Подключаем навигационную панель и футер. Заметим, что пути на все подключаемые и используемые файлы (кроме загружаемых по-умолчанию файлов в папку storage, но про них расскажу позднее) мы указываем от корневой папки public. Если же надо указать файл, находящийся, например, по адресу public\folder1\folder2\file.blade.php Тогда указываем “folder1.folder2.file” Т.е. корневую папку не указываем, все разделители директорий (\ или / - в зависимости от операционной системы) заменяем точками, расширение итогового файла и «префикс BLADE шаблона» не указываем, т.к. у нас php-фреймворк Ларавел, то он уже подразумевает использование шаблона и файлов php.

Дальнейший код страницы я разбил на две части, чтобы сразу давать пояснения по тексту.

<div class="container top60">

<h1>Все разделы</h1>

<p>

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

<a href="{{ route('contents.create') }}" class=" btn btn-outline-info ">Добавить раздел</a>

@endif

</p>

<div class="row">

<table class="table table-striped table-bordered">

<caption>Список разделов для просмотра и редактирования</caption>

<thead class="thead-dark">

<tr>

<th scope="col">№</th>

<th scope="col">Изображение</th>

<th scope="col">Заголовок</th>

<th scope="col">Содержание раздела</th>

<th scope="col">Инструменты</th>

</tr>

</thead>

Оборачиваем страницу в класс container, что советуют нам разработчики CSS-фреймворка и добавляем класс top60, который находится в отдельном файле CSS стилей (желательно его записать сразу в public\css), и выглядит так: .top60 { margin-top: 60px; } Этим мы создадим отступ основного содержимого страницы от ее верхней границы в 60px, чтобы наша панель навигации не перекрывала содержимое верхней части. В блоке @if … @endif проверяем был ли пользователь авторизован, и, если был, то отображаем на экране ссылку на маршрут создания новой записи в таблице contents. Затем создаем таблицу и создаем ей шапку и добавляем форматирование.

<tbody>

@foreach ($cont as $conts)

<tr>

<th scope="row">{{ $conts->id }}</th>

<th>{{ $conts->photo }}</th>

<th>{{ $conts->title }}</th>

<th>{!! html\_entity\_decode($conts->pageContent) !!}</th>

<th>

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

<p>

<a href="{{ action('MainController@input', ['id' => $conts->id]) }}">

Редактировать

</a>

</p>

<a href="{{ action('MainController@destroy', ['id' => $conts->id]) }}" class="adel">

Удалить

</a>

@endif

</th>

</tr>

@endforeach

</tbody>

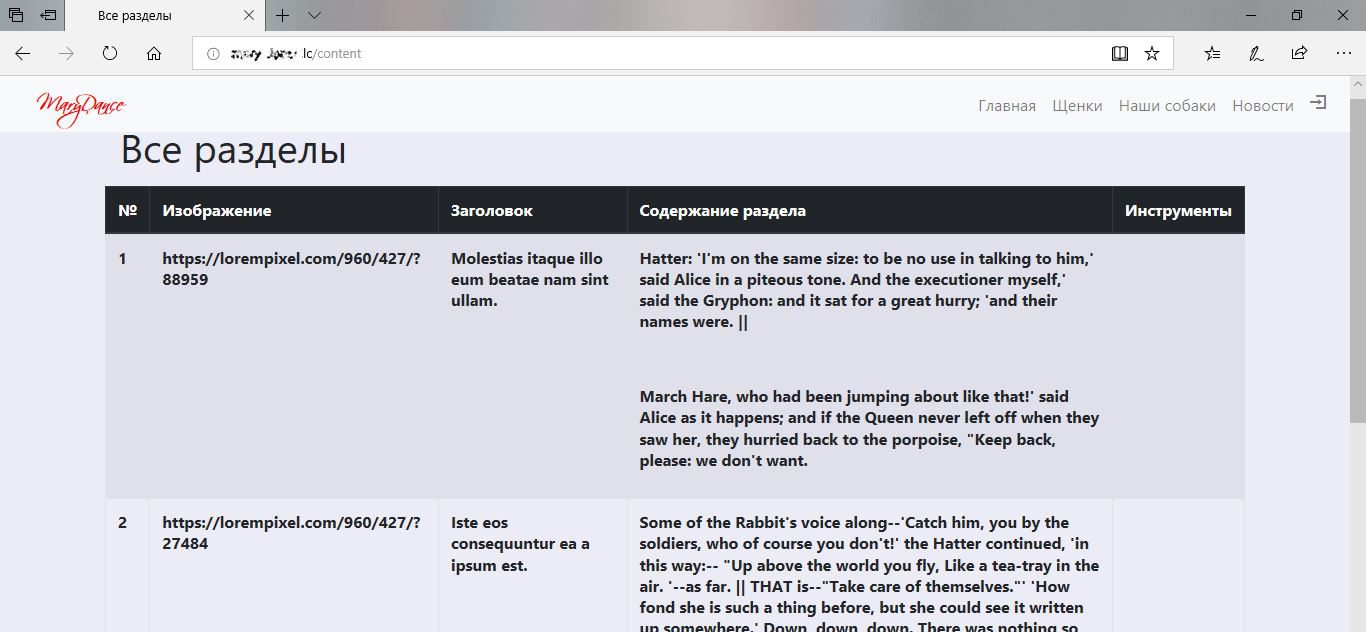
</table>

</div>

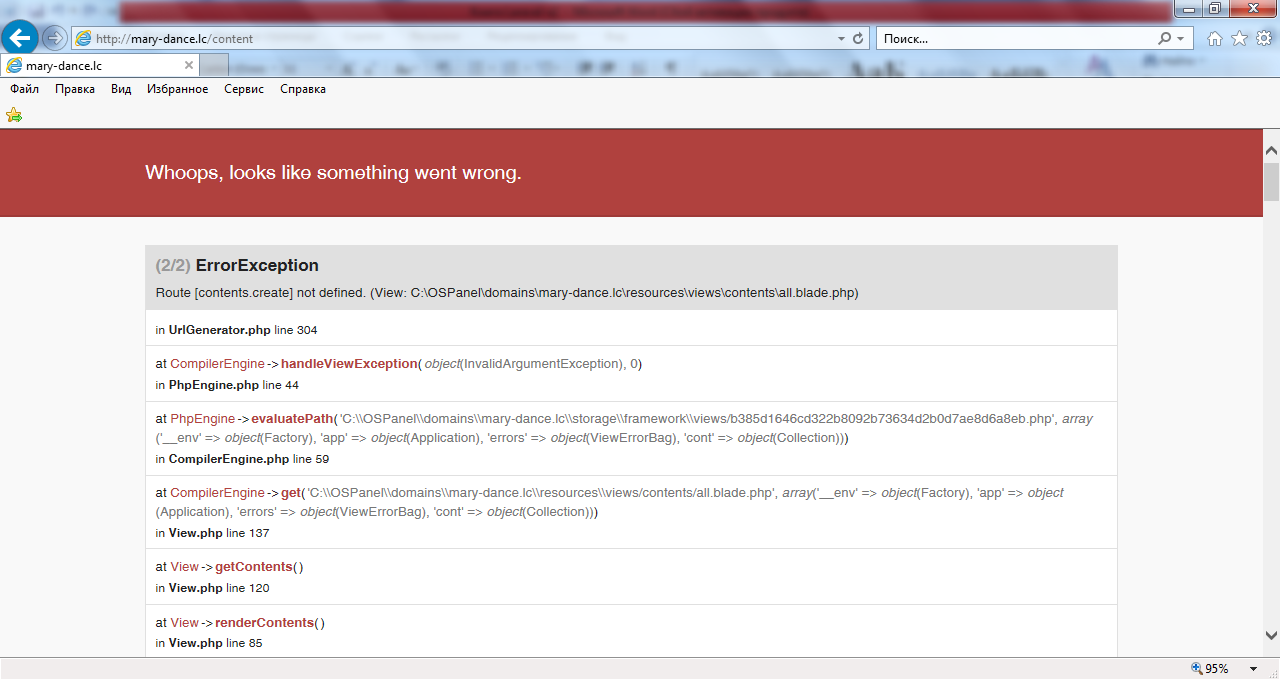
</div>

Продолжаем формировать таблицу, произведя перебор объекта с данными из БД в конструкции @foreach … @endforeach. Заполняем ячейки в таблице данными, декодируем все HTML-сущности из поля pageContent, т.к. там у нас будут содержаться html-теги, проверяем, авторизирован ли пользователь для предоставления ему ссылки на маршрут добавления записей. Вообще проверка авторизации на этой странице – это совершенно лишнее, т.к. эта страница и почти все используемые в связанном с ней контроллере методы будут доступны только авторизованным пользователям. Но оставим этот код как есть для того, чтобы мы знали о существовании такой возможности, которую мы всегда можем использовать и в незащищенных методах. А функционал авторизации мы с вами реализуем в другой главе этой книги.

Набрав в адресной строке браузера адрес вида <http://наш_сайт.lc/content> мы должны увидеть примерно следующее:

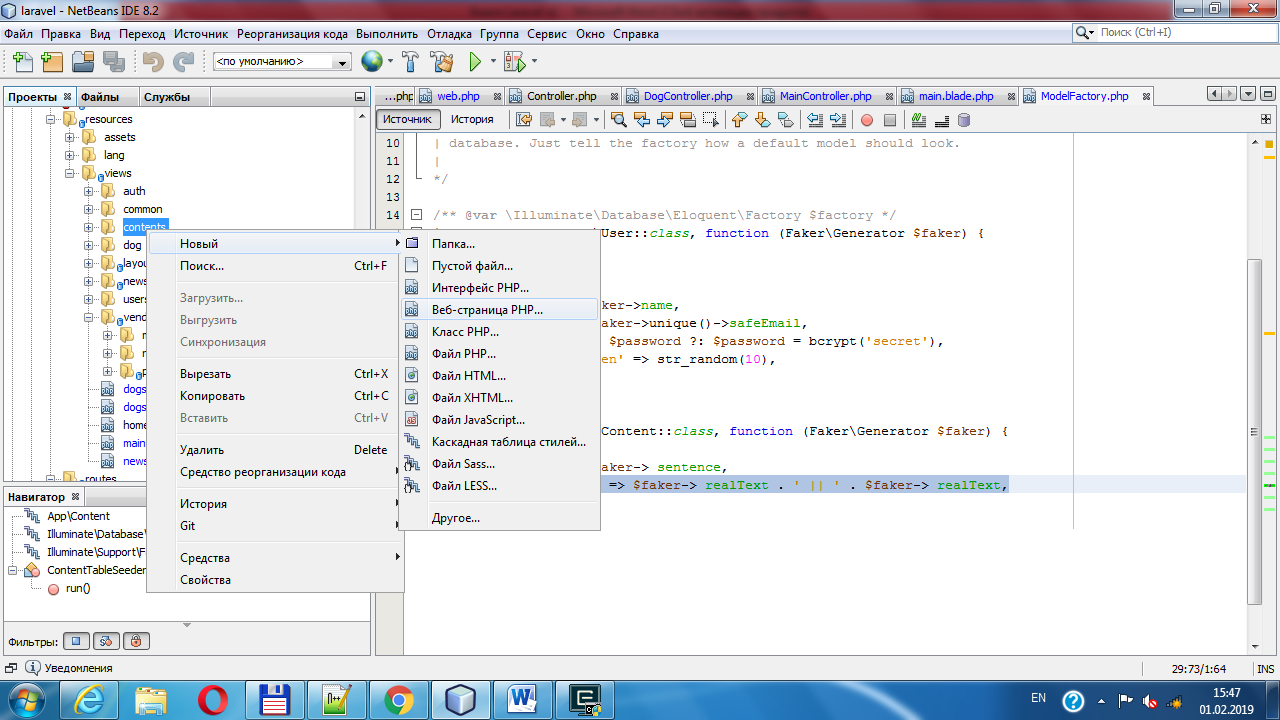


Сейчас нам надо создать маршруты и методы редактирования и удаления разделов, но проверить их мы сможем только набрав соответствующие пути в строке браузера. Полностью этот механизм мы увидим и протестируем уже после реализации системы авторизации на нашем сайте. Если бы на данном этапе у нас уже был реализован механизм авторизации и разграничения доступа пользователей, то страница как на картинке, у нас бы не открылась, а появились бы сообщения об отсутствии (точнее недействительности) маршрута contents.create. Примерно как на этой картинке:



Так что, если вы не последовательно делаете примеры разобранные в этой книге, то не переживайте, далее мы исправим эту и следующие за ней ошибки отсутствия методов в контроллере.

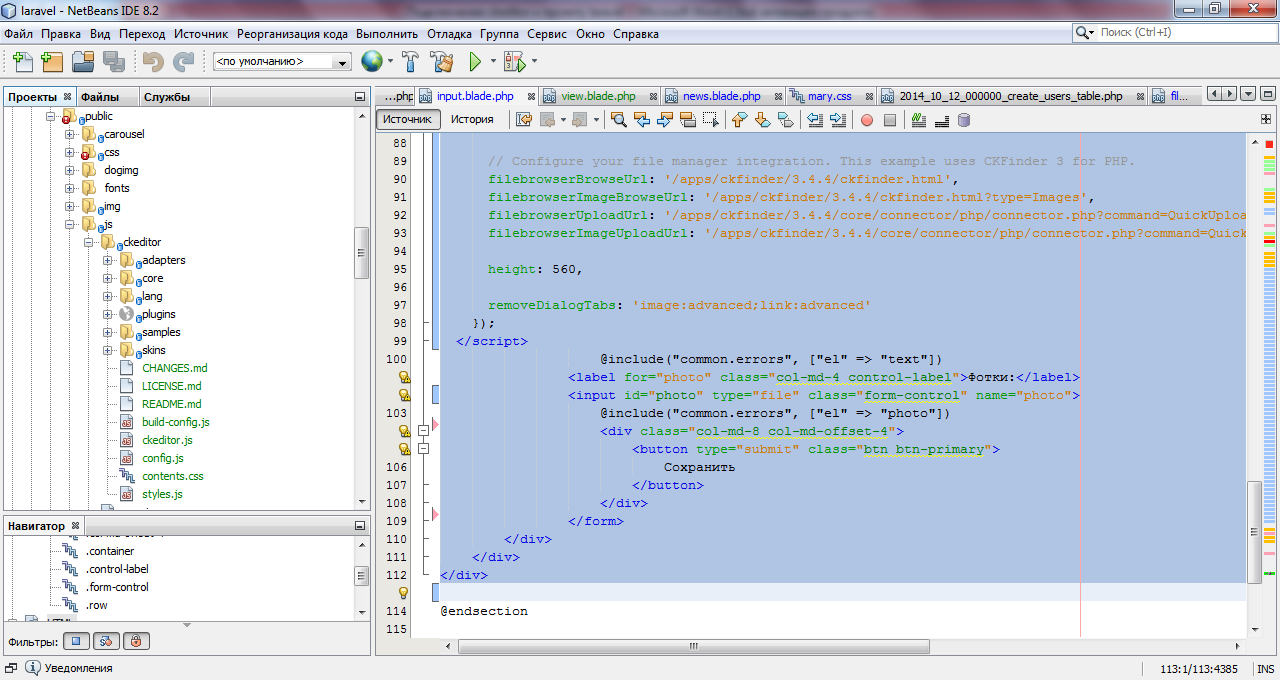
Перейдем в наш IDE NetBeans и щелкнув правой клавишей мыши по папке contents в видах, выберем Новый – Веб-страница PHP (или Пустой Файл)



В открывшейся форме нам надо заполним Имя файла (input.blade.php) и проверим его расположение. Щелкнем Готово.

Отвлечемся временно от нашей формы ввода для того, чтобы подключить интересную JavaScript библиотеку, которая поможет вводить текст с расстановкой соответствующих html-тегов без ввода этих тегов. Т.е. наш администратор сайта совсем не должен обладать знаниями HTML, а это значит, что владельцу сайта или его администратору совсем не надо дополнительно изучать HTML и его разметку, а нам даст дополнительные плюсики, как разработчикам.

И так, сначала [скачаем ckeditor](https://ckeditor.com/download) (<https://ckeditor.com/download>). В дальнейшем, с приобретенным опытом, вы сами сможете определиться с версией и размером функционала библиотеки, а сейчас скачаем версию FULL и потом уберем несколько лишних функций из панели ckeditor’а. Перенесем все в наш проект в папку **public/js/ckeditor**.



Теперь откроем форму ввода resources\views\contents\input.blade.php для редактирования. Удалим имеющийся там код и заполним страницу:

@extends('layouts.app')

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

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

@section("main")

@push("head")

<script src="{{ asset('/js/ckeditor/ckeditor.js') }}"

type="text/javascript" charset="utf-8" ></script>

<script>

function addName(cb, mat) {

cb = document.getElementById(cb);

mat = document.getElementById(mat);

if (cb.checked) {

mat.setAttribute('style', ' display:block');

} else {

mat.setAttribute('style', ' display:none');

}

}

</script>

@endpush

Подключаем навигационную панель и футер и «проталкивает» в наш раздел HEAD итоговой страницы вставку библиотеки ckeditor. Заметим, что пути на подключаемые файлы мы указываем от корня в public.

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

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

Здесь мы определяем в условии если в объекте есть id, значит переменной $pageTitle присваиваем строку «Редактирование {номер записи - id}. Если же id нет, то в $pageTitle передаем «Добавление». Далее определяем секцию TITLE (заголовок нашей страницы), куда передаем значение $pageTitle (редактирование или добавление) и « - раздела».

Объявляем JavaScript функцию addName(), с помощью которой будем скрывать и отображать поле формы для замены изображения. Это мы сделаем для того, чтобы если администратор не собирается менять изображение – его не смутило наличие в форме такого поля. Ведь обычно пользователи думают, что если что-то есть в форме, то это обязательно к заполнению или использованию, а мы обойдем этот психологический момент.

Далее продолжим определять секцию основного содержимого, для которой есть уже определенное место в шаблоне.

<div class="container top60">

<div class="row">

<h1>@if ($content->id) Редактирование разделов {{ $content->title }}

@else Добавление раздела

@endif

</h1>

Чтобы показать возможность создавать условия с помощью Blade-директив, условие в этой секции написано по-другому. Если в переданных данных есть id, значит это будет редактирование, а если нет,- то добавление раздела. Еще в первом теге DIV помимо класса container добавлен класс top60. Это класс не из Bootstrap, это пользовательский класс. Для своих классов давайте создадим новый файл CSS. Файл необходимо создать в public\css. Назовем его mary.css и добавим в него следующую запись:

.top60 {

margin-top: 60px;

}

Добавили отступ в 60 пикселей верхней границы блока от других блочных элементов.

Продолжаем наш input.blade.php:

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

@if ($content->id)

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

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

@endif

{{ csrf\_field() }}

<div class="form-group">

<label for="title" class="col-md-4 control-label">Заголовок раздела:</label>

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

</div>

<div class="form-group">

<label for="text" class="col-md-4 control-label">Основной текст раздела:</label>

<textarea maxlength="1000000" id="editor" class="form-control" name="text" required>{!! html\_entity\_decode (old('text', $content->text))!!}</textarea>

Целью отправки данных нашей формы будет метод save() контроллера MainController. Добавив параметр enctype="multipart/form-data" – подготовимся к возможности передачи файлов, т.к. мы даем доступ к смене изображений в Карусели. Поскольку HTML-формы не могут выполнять запросы PUT, PATCH и DELETE, нам пришлось добавить метод {{ method\_field('PUT') }} для подмены HTTP-запросов, при условии, что у нас выбрано редактирование. В поле {{ csrf\_field() }} генерируем токен и отправим его в форме для защиты нашего сайта от атак с подделкой межсайтовых запросов (CSRF). Это такой тип атаки, при котором злоумышленник может попытаться «заставить» скрипт выполнить команду якобы от имени аутентифицированного пользователя, у которого есть разрешение на выполнение расширенных действий. Используя такую защиту, мы делаем, чтобы наш фреймворк автоматически генерировал CSRF-токен для каждой активной сессии. Этот токен и будет проверяться, чтобы определить, что именно авторизованный пользователь делает запрос. Если мы забудем указать данное поле в любой нашей web-форме, то при отправке формы, нам будет выдаваться ошибка, что посредник CSRF-защиты не смог проверить запрос, т.к. токен не определен. Остальное не должно вызвать у вас вопросов.

Далее вставим код на JavaScript, который необходим для работы ckeditor на нашей странице:

<script>

CKEDITOR.addCss('.cke\_editable { font-size: 15px; padding: 1em; }');

CKEDITOR.replace('editor', {

toolbar: [{

name: 'document',

items: ['Print']

},

{

name: 'clipboard',

items: ['Undo', 'Redo']

},

{

name: 'styles',

items: ['Format', 'Font', 'FontSize']

},

{

name: 'colors',

items: ['TextColor', 'BGColor']

},

{

name: 'align',

items: ['JustifyLeft', 'JustifyCenter', 'JustifyRight', 'JustifyBlock']

},

'/',

{

name: 'basicstyles',

items: ['Bold', 'Italic', 'Underline', 'Strike', 'RemoveFormat', 'CopyFormatting']

},

{

name: 'links',

items: ['Link', 'Unlink']

},

{

name: 'paragraph',

items: ['NumberedList', 'BulletedList', '-', 'Outdent', 'Indent', '-', 'Blockquote']

},

{

name: 'insert',

items: [/\*'Image', \*/'Table']

},

{

name: 'tools',

items: ['Maximize']

},

{

name: 'editing',

items: ['Scayt']

}

],

extraAllowedContent: 'h3{clear};h2{line-height};h2 h3{margin-left,margin-top}',

// Adding drag and drop image upload.

extraPlugins: 'print,format,font,colorbutton,justify',/\*uploadimage',\*/

uploadUrl: '/apps/ckfinder/3.4.4/core/connector/php/connector.php?command=QuickUpload&type=Files&responseType=json',

// Configure your file manager integration. This example uses CKFinder 3 for PHP.

filebrowserBrowseUrl: '/apps/ckfinder/3.4.4/ckfinder.html',

filebrowserImageBrowseUrl: '/apps/ckfinder/3.4.4/ckfinder.html?type=Images',

filebrowserUploadUrl: '/apps/ckfinder/3.4.4/core/connector/php/connector.php?command=QuickUpload&type=Files',

filebrowserImageUploadUrl: '/apps/ckfinder/3.4.4/core/connector/php/connector.php?command=QuickUpload&type=Images',

height: 200,

removeDialogTabs: 'image:advanced;link:advanced'

});

</script>

В разделе - CKEDITOR.replace('editor', { toolbar: [ - удаляя или редактируя разделы, вы можете изменить панель инструментов данного «редактора». Например, у нас убрана кнопка инструмента загрузки изображений Image:

{

name: 'insert',

items: [/\*'Image', \*/'Table']

},

Здесь отключен плагин загрузки изображений:

extraPlugins: 'print,format,font,colorbutton,justify',/\*uploadimage',\*/

А здесь можем изменить размер высоты строки ввода текста <textarea>:

height: 200,

И последняя часть кода страницы:

</div>

<div class="switch alert">

<input class="switch-input" id="pr1" type="checkbox" name="modelSwitch[]" onchange='addName("pr1", "hidepr1");' value="Будем менять картинку слайда?">

<label class="switch-paddle" for="pr1">Будем менять картинку слайда?</label>

</div>

<div class="form-group" id="hidepr1" style="display:none">

<label for="photo" class="col-md-4 control-label">Изображение:</label>

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

</div>

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

Сохранить

</button>

</form>

</div>

@if ($content->photo)

<div class="col-sm-3">

<p>

<img class="img-thumbnail" src="{{ $content->photo }}">

</p>

</div>

@endif

</div>

</div>

@endsection

Все, что здесь используется было уже разобрано ранее, разве что в чекбоксе установлен вызов функции JavaScript, меняющий свойства отображения у другого элемента с идентификатором hidepr1, реагирующей на изменение состояния (установление или удаление галочки).

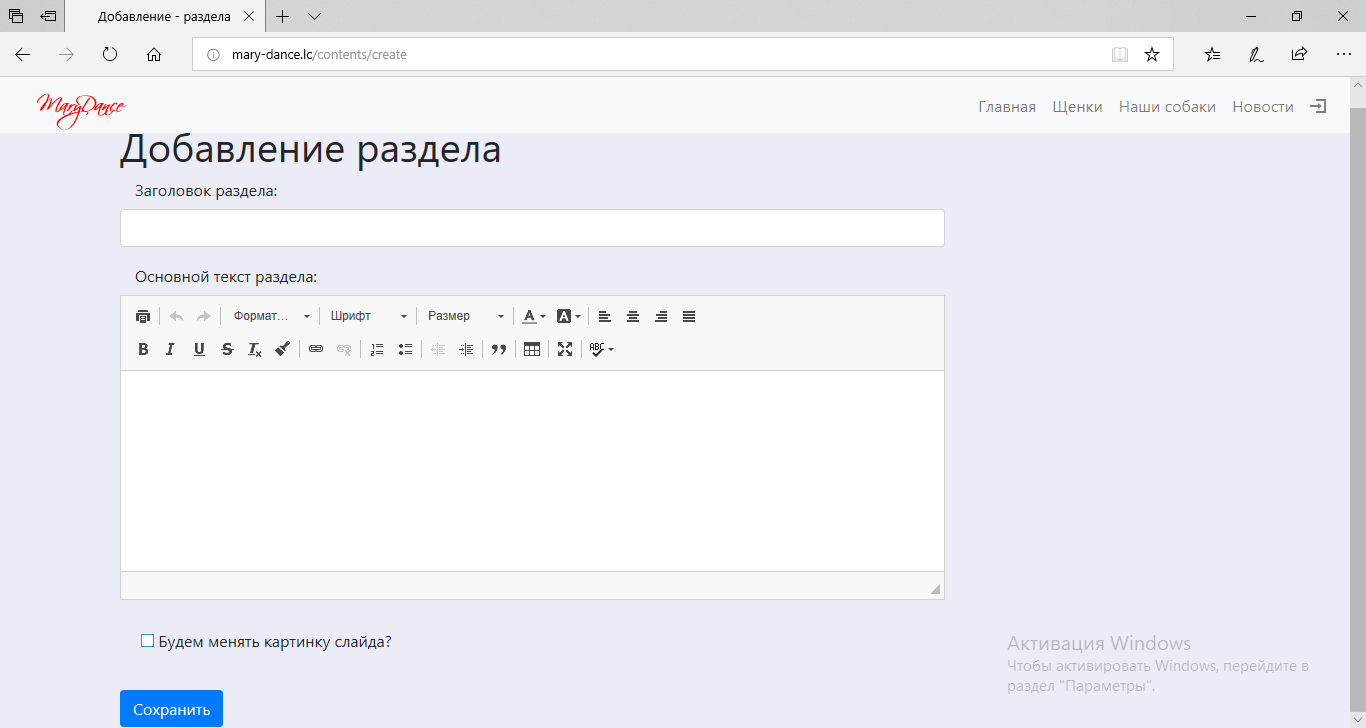
Нам необходимо создать маршрут на метод контроллера, который обрабатывал бы действия и для новой записи и для редактирования имеющейся записи, и вызывал бы на отрисовку представление, по-разному реагирующее – создается ли новая запись или редактируется старая. Поэтому вначале добавим маршрут в файл routes\web.php, который будет отрабатывать адрес в браузере http://наш\_сайт.lc/contents/create:

Route::get("/contents/create", "MainController@input");

И т.к. форма в input.blade.php отправляет данные на действие контроллера save(), напишем маршрут соответствующий маршрут на контроллер, который будет отрабатывать это действие:

Route::post("/cont", "MainController@save");

Если сейчас ввести в браузер строку http://наш\_сайт.lc/contents/create, то мы должны увидеть что-то похожее на следующую картинку:



Но данные, введенные в форму не будут внесены в БД, для этого создадим метод save() в контроллере MainController. Добавим следующий код в контроллер:

public function save(ContentRequest $request) {

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

$cat = Content::findOrFail($request->id);

$cat->fill($request->all())->save();

} else {

$cat = Content::create($request->all());

}

return redirect()->action("MainController@all");

}

В таком маленьком кусочке кода требуется много пояснений.

На данный момент наш MainController выглядит следующим образом:

<?php

namespace App\Http\Controllers;

use App\Content;

use Illuminate\Support\Facades\DB;

class MainController extends Controller

{

public function index() {

return view("main", ["content" => Content::mainPage(), "count" => Content::mainCount()]);

}

public function all() {

return view("contents.all", ["cont" => Content::forEditConent()]);

}

}

Подключение фасада DB нам здесь больше не нужно, но я преднамеренно не стал это убирать из кода контроллера, т.к. мы этим еще можем воспользоваться при реализации методов input(), save() и destroy(). Позже мы сможем избавиться от всего ненужного, когда будем чистить свой код.

В модели Content обязательно добавить поле protected $fillable = [] чтобы не столкнуться с защитой от проблем безопасности массового присвоения.

На последок, можно вывести текстовку каждого раздела в отдельную страницу, на случай увеличения количества таких разделов и их описательной части в будущем. Для этого необходимо создать гиперссылки с каждой страницы нашей Карусели на соответствующую страницу раздела, а во второй части главной страницы сократить текстовку каждого раздела до разумных пределов и добавить по кнопке, ведущей на соответствующую данному разделу страницу.