Теперь мы создаем страницу просмотра разделов главной страницы 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() {

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

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

}

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 точно 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

{

protected $fillable = ["id", "title", "pageContent" **, "photo"**];

protected $primaryKey = "id";

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();**

**}**

}

Этим мы 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),**

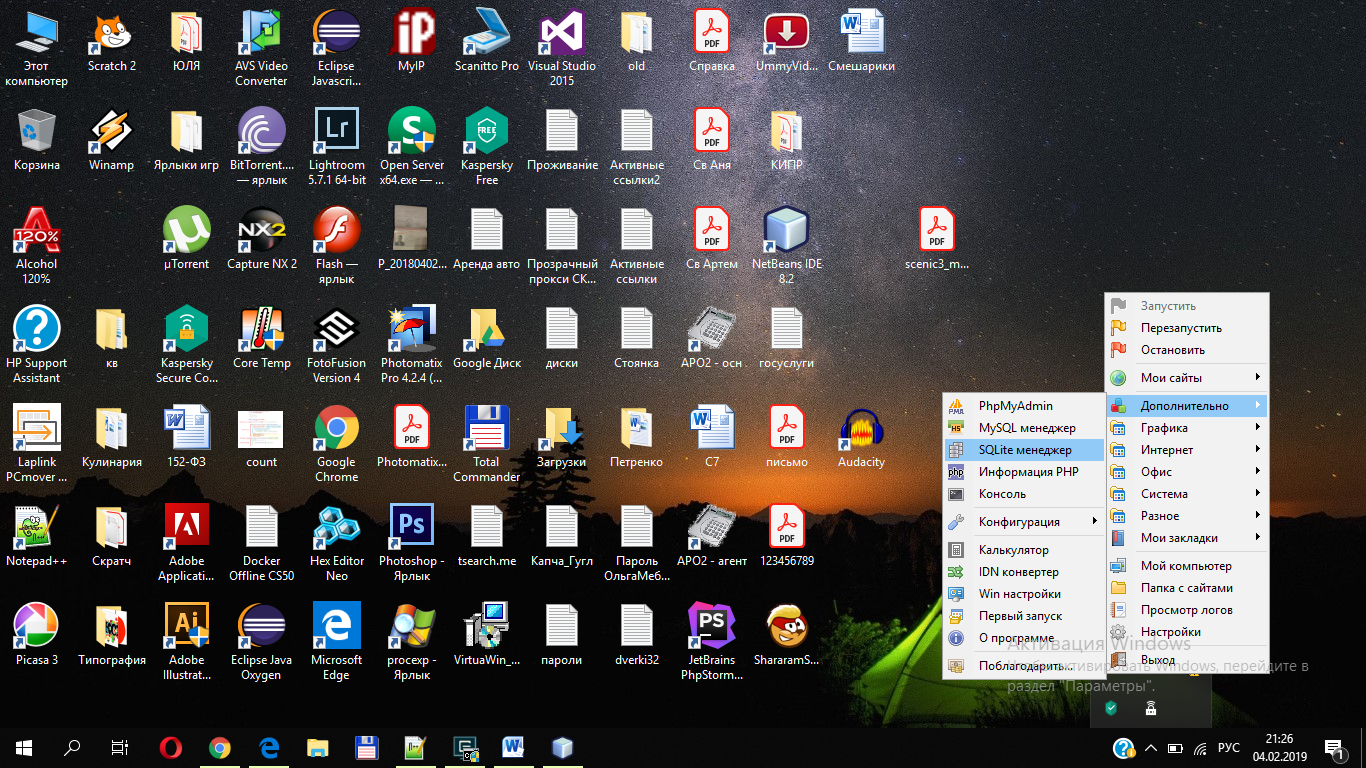
];

});

database\seeds\DatabaseSeeder.php оставляем без изменений, 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.

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(); предоставит нам эти данные. После внесения изменений эти методы 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. Стоит заметить, что в этих переменных используются разные типы данных. В $content – коллекция объектов (далее коллекция) этот тип возвращает метод get(); а в $count – число – тип integer.

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.

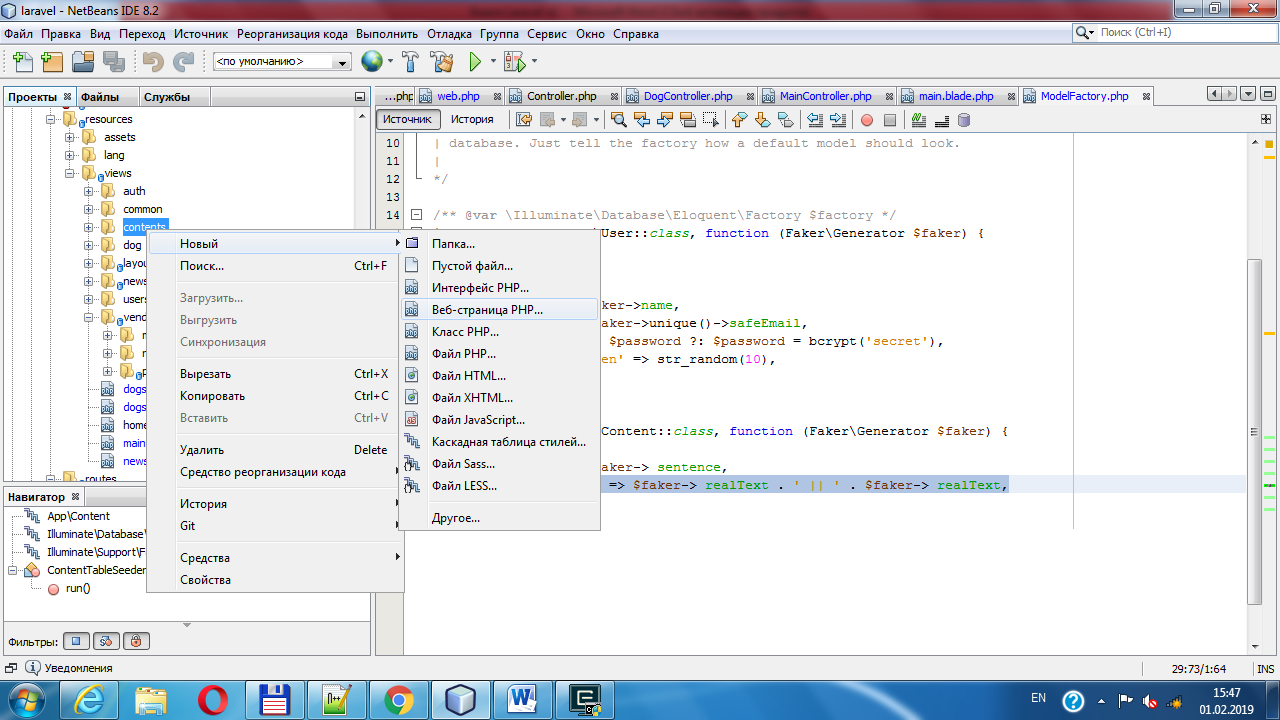
Запускаем Консоль OpenServser’a как мы это делали ранее, и если мы находимся в корне папки OpenServer, то переходим в папку с представлениями нашего проекта:

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. Щелкаем 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, шапку таблицы 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>

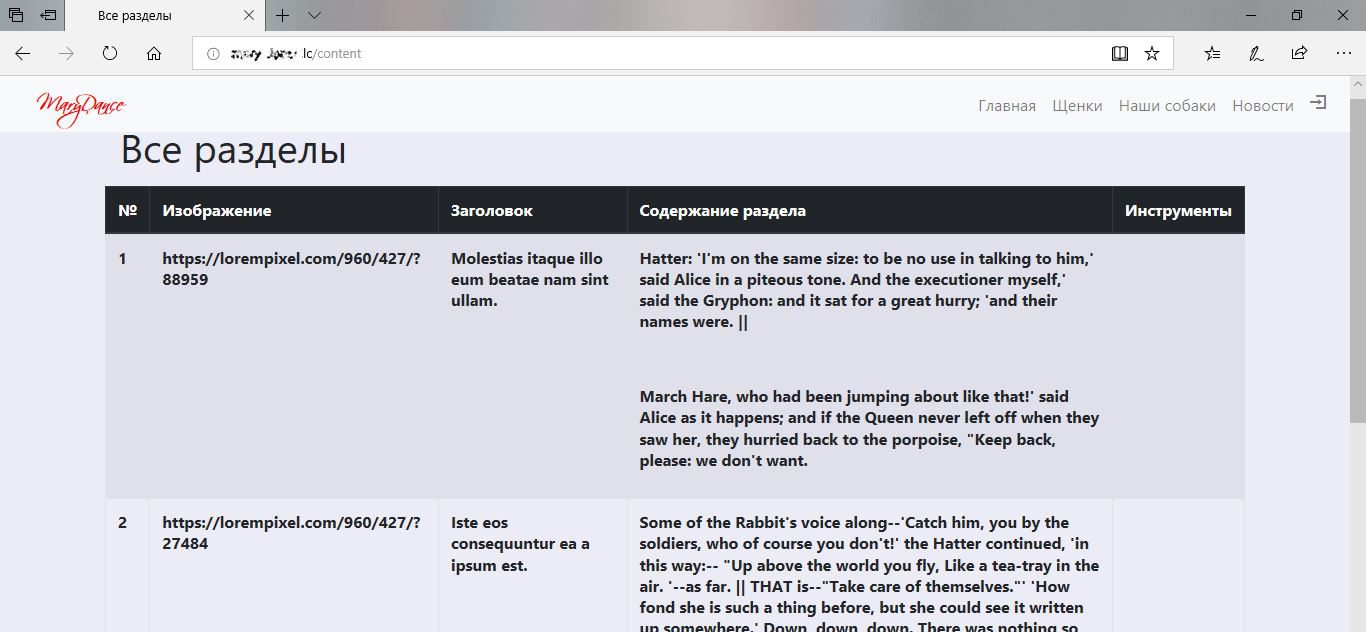
</table>

</div>

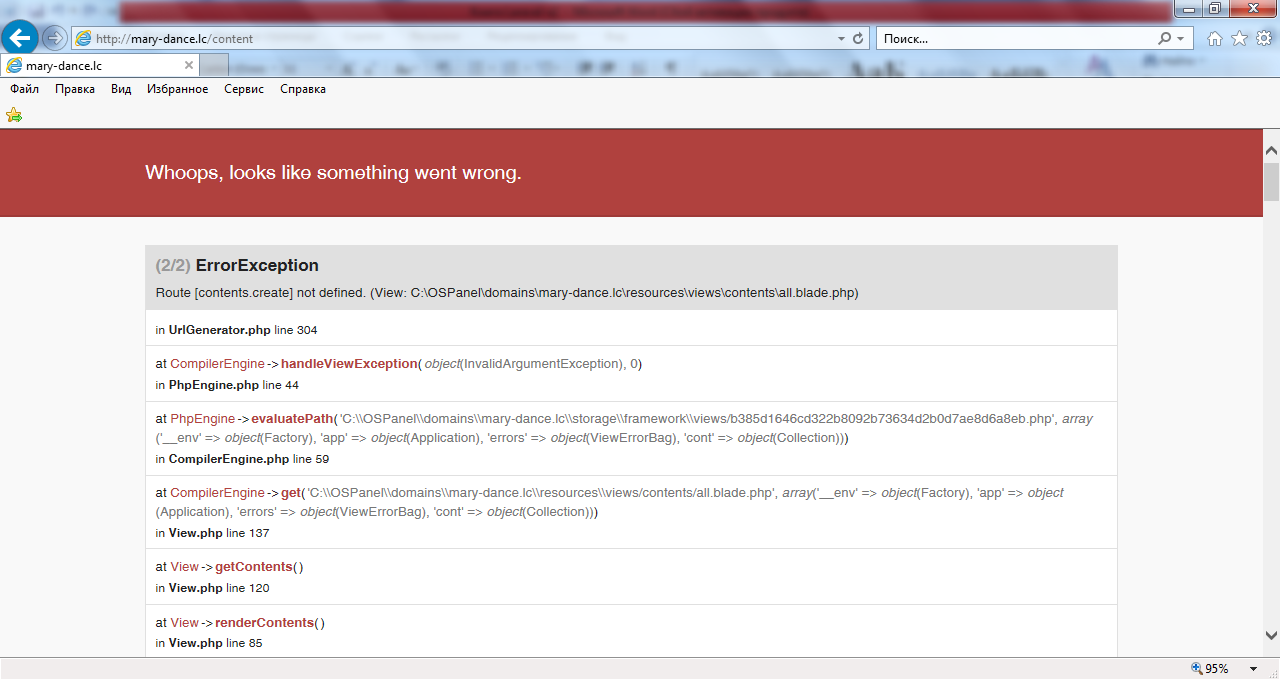
</div>

Продолжаем формировать таблицу, произведя перебор коллекции 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 отображения ему ссылок на маршруты редактирования и удаления записей. Вообще проверка авторизации на этой странице – это совершенно лишнее, т.к. эта страница и почти все используемые в связанном с ней контроллере методы мы сделаем доступными только авторизированным 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 появилось бы сообщение об отсутствии (недействительности) маршрута или метода контроллера. 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.

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");

Добавим в контроллер метод input(), ссылку на который мы указали для действия «Редактировать». Допишем следующее в наш файл контроллера MainController:

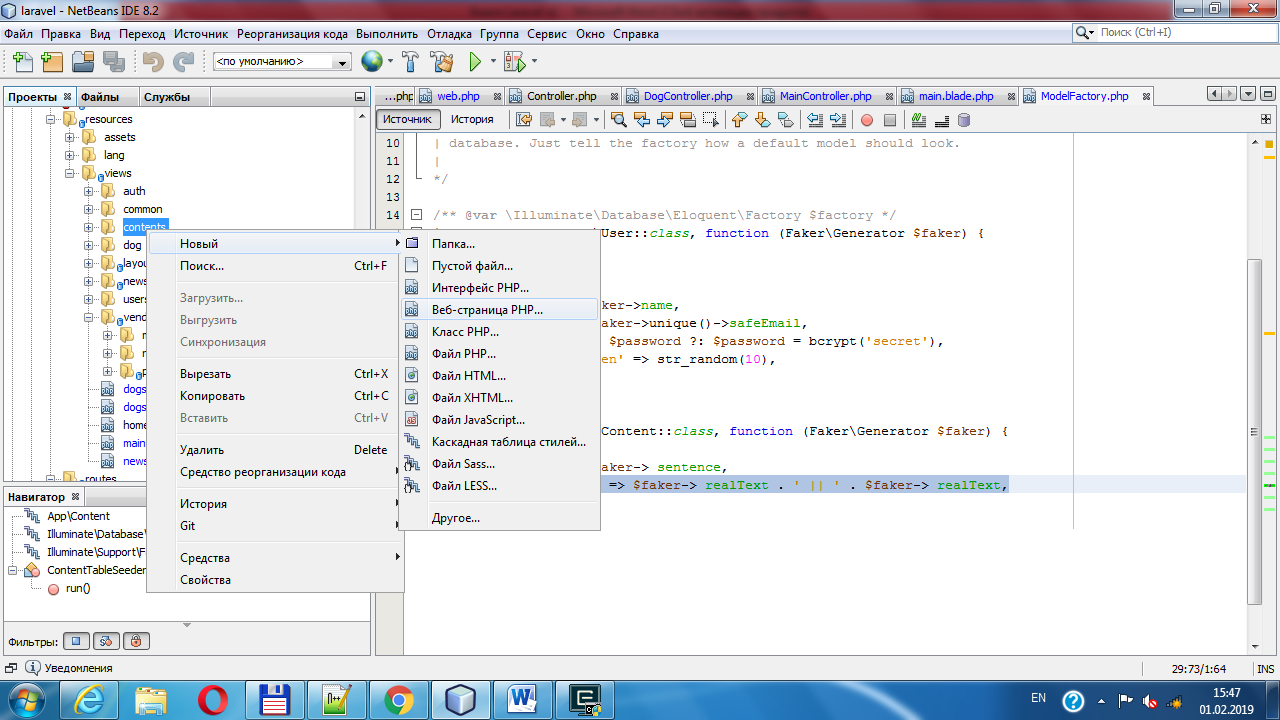
public function input(Content $content) {

return view ("contents.input", ["content" => $content]);

}

Мы передадим в представление resourses\views\contents\input.php данные из модели Content.php, можно сказать, что мы внедрили модель в наш контроллер. Если в данных будет id, то модель вернем нам конкретную запись для редактирования. Если же id не будет, то модель не вернет ничего,- значит мы занимаемся созданием новой записи.

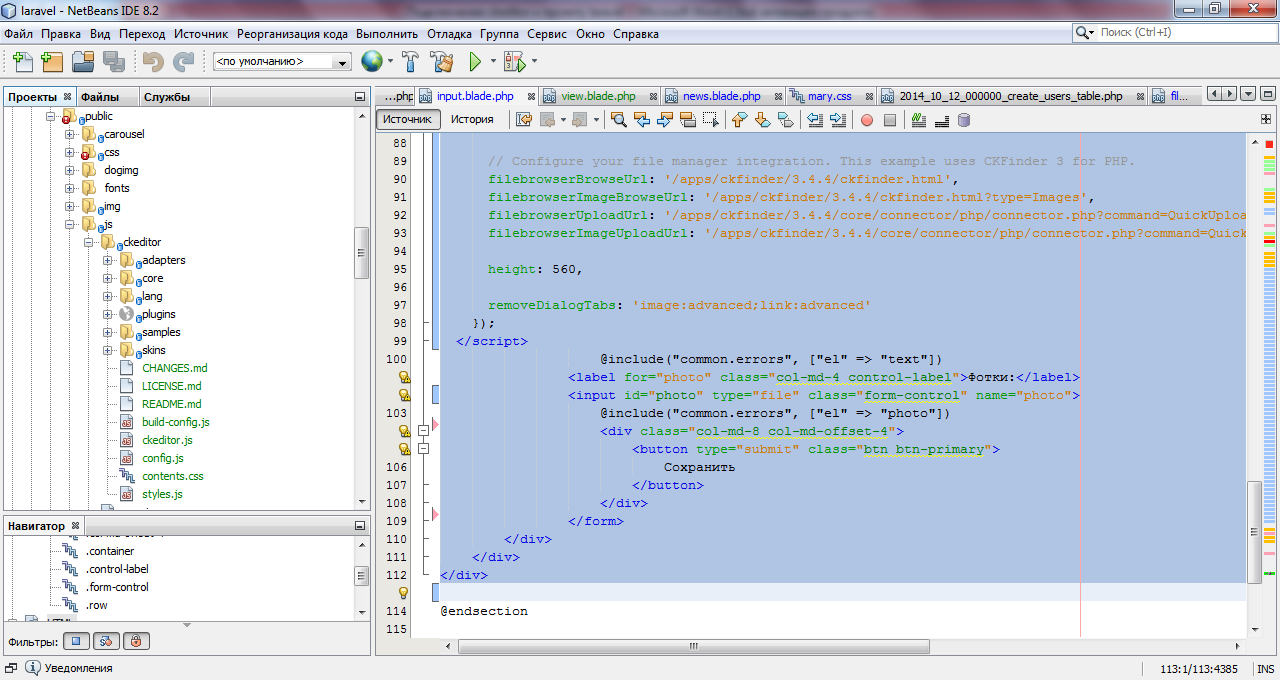
Создадим шаблон представления для создания и редактирования записей о разделах главной страницы. Перейдем в наш IDE NetBeans и щелкнув правой клавишей мыши по папке contents в видах, выберем Новый – Веб-страница PHP (или Пустой Файл)



В открывшейся форме нам надо заполнить Имя файла (input.blade.php) и проверить 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.

<?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» конкатенацией строк.

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 свое 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;

}

Этим мы добавили отступ в 60 пикселей верхней границы блока от других блочных элементов. Нам надо добавить ссылку на этот файл в resources\views\layouts\app.blade.php. Добавим эту строку сразу же после подключения css-файлов Bootstrap:

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

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>

Метод old() извлекает старое значение поля формы, введенное в сессии.

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 отправляем его из формы с данными 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).

Создадим сервисный класс запроса ContentRequest для данных отправляемых формой input.blade.php

Для этого откроем Консоль, перейдем в папку нашего проекта и выполним следующую команду:

php artisan make:request ContentRequest

У нас создастся файл app\Http\Requests\ContentRequest.php. Изменим в нем код на:

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContentRequest extends FormRequest

{

public function authorize()

{

return true;

}

public function rules()

{

return [

'title' => 'required|max:200',

'pageContent' => 'required|max:1000000',

];

}

public function messages() {

return [

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

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

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

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

];

}

}

Мы объявляем (class ContentRequest extends FormRequest) что класс ContentRequest наследуется от класса FormRequest, получая его функционал и использует еще и свои методы класса, которые мы сами и пропишем. В методе rules() прописываем правила для полей формы. В нашем примере указываем, что поля title и pageContent являются обязательными для заполнения (required) и добавляем им ограничения максимально разрешенного количества символов для ввода. Можем через знак | указать другие правила для полей формы.

В методе message() меняем стандартные сообщения об ошибках ввода в поля формы на сообщения, которые мы сами пропишем для каждого действия.

Так как форма в input.blade.php отправляет данные на действие контроллера save(), напишем соответствующий маршрут на контроллер, который будет отрабатывать действие создания новой записи, отправленное методом POST:

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

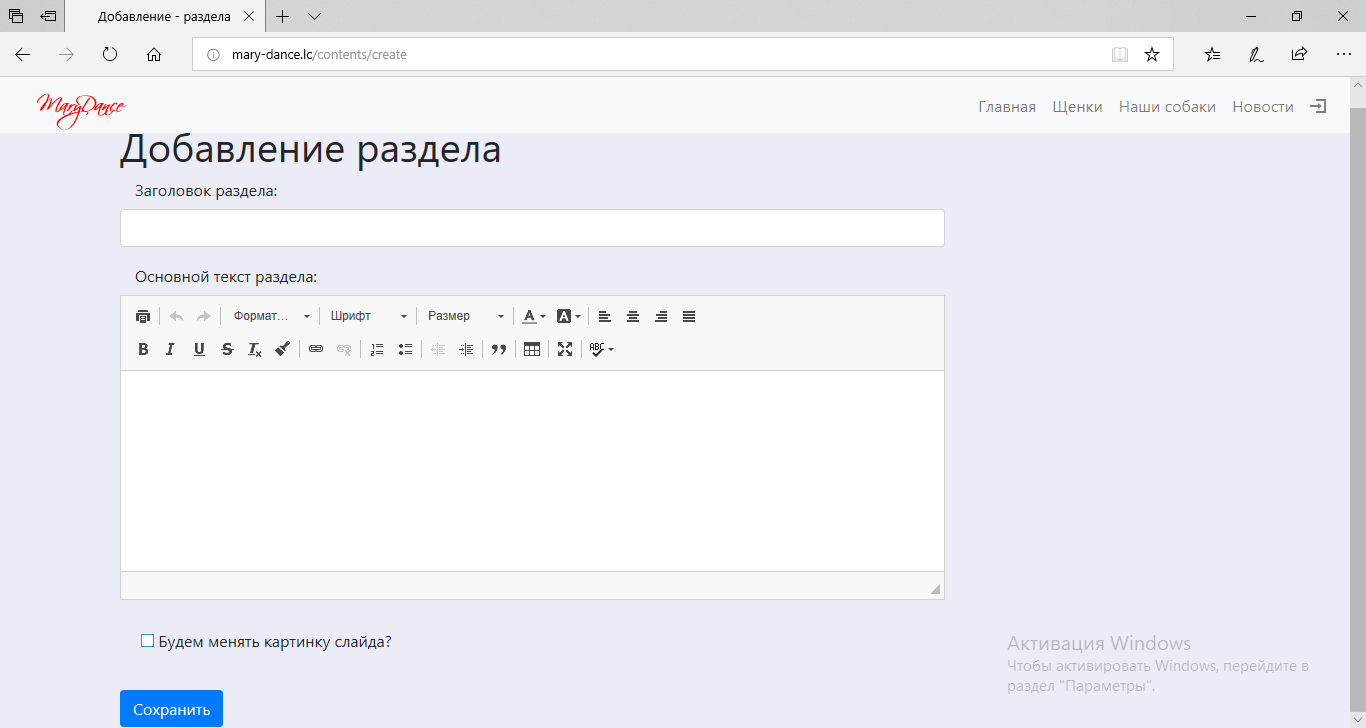
Эту же форму мы будем использовать для редактирования существующей записи, но отправлять эти данные мы будем методом PUT. Поэтому напишем маршрут и для этого действия:

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

Мы можем объединить два этих маршрута в один с использованием метода match() фасада Route. Удалим два написанных маршрута и на их месте напишем следующий:

Route::match(["post","put"],"/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, отправленные формой сейчас 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.

public function save(ContentRequest $request)

Сразу надо оговориться, что если мы внедряем запрос ContentRequest как показано выше, то необходимо сразу после указания пространства имен данного класса добавить использование запроса с полным путем от корня:

use App\Http\Requests\ContentRequest;

Если мы не добавим эту строку, то внедрение запроса в контроллере выглядело бы следующим образом: public function save(\App\Http\Requests\ContentRequest $request) и везде в контроллере, при указании этого запроса, нам надо было бы указывать его полный путь \App\Http\Requests\ContentRequest. А так, указав его пространство имен один раз в разделе объявлений, мы везде можем обращаться к запросу и указывать его просто – ContentRequest.

Итак, мы делаем получение экземпляра текущего HTTP-запроса через внедрение зависимости. Экземпляр входящего запроса будет автоматически внедрён сервис-контейнером. Сервис-контейнер в Laravel — это мощное средство для управления зависимостями классов и внедрения зависимостей. Внедрение зависимостей — это термин, который означает что мы можем получить данные из источника данных прямо в этот класс через его конструктор (public function \_\_construct()) или метод. Т.е. мы внедряем сервис, который умеет получать данные, поступившие из формы ввода, заодно он может провести валидацию введенных данных, установить правила для ввода данных, изменить сообщения об ошибках, возникающих при неверном вводе данных и т.п. Этот класс относится к так называемым сервис-классам Form Requests. Это специальные классы HTTP-запроса, содержащие в себе логику валидации. Они обрабатывают запрос до того, как он поступит в контроллер.

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

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

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

Если в массиве, переданном из формы input.blade.php и принятым в переменную $request есть поле id, значит мы имеем запрос на изменение записи. Мы можем в модели Content написать свой метод на поиск записи, но наша же модель наследуется от класса Model, значит мы можем пользоваться методами этого класса. Поэтому, обращаясь к модели Content, мы вызываем метод findOrFail() класса Model. Метод findOrFail() получает первый результат запроса, но так как у нас одна запись, то именно ее метод нам и найдет. И это будет объект записи. Ну а если же результат не найден, происходит исключение Illuminate\Database\Eloquent\ModelNotFoundException. Если исключение не поймано, пользователю автоматически посылается HTTP-отклик 404. Нет необходимости писать явные проверки для возврата откликов 404 при использовании этого метода. Затем мы все значения массива в $request присваиваем полям объекта записи и сохраняем изменения в нашу таблицу БД.

} else {

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

}

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

}

Если же в массиве $request нет поля id, значит это создание новой записи. Используем через модель Content метод класса Model создания записи – create() и передаем весь массив $request. Метод create() создаст нам новую запись со следующим id и расставит сам все ключи и значения. В итоге, после редактирования или создания записи, вызываем перенаправление на действие метода контроллера MainController all(), который мы уже написали и знаем, как он работает.

Нам осталось написать только два маршрута: для редактирования и для удаления записей, а также создать метод в MainController, который и будет производить удаление записей. Маршруты в routes\web.php напишем такие:

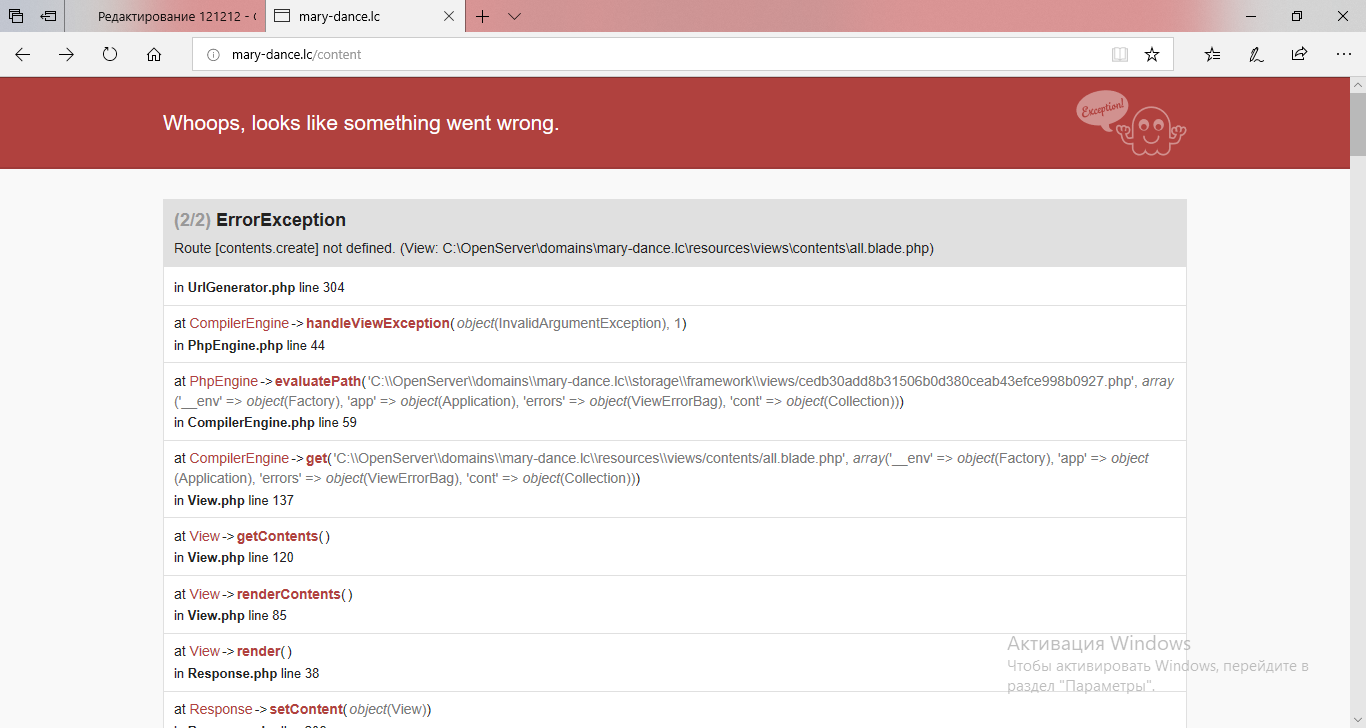
- для редактирования:

Route::get("/cont/{content}/edit", "MainController@input");

- для удаления:

Route::get("/cont/{id}/delete", "MainController@destroy");

Однако, если попытаться перейти по адресу http://наш\_сайт.lc/content, что вызовет на отрисовку файл resources\views\contents\all.blade.php, появится ошибка:



о том, что маршрут contents.create не определен. Избавится от этой ошибки мы можем, изменив маршрут для действия create, сделав его именованным. Именованные маршруты позволяют удобно генерировать URL-адреса и делать переадресацию на конкретный маршрут. Если бы мы сделали ссылку на добавление записи через action, то у нас ссылки добавления и редактирования записей формировались бы вида contents/create, что не дало бы нам производить редактирование записей.

Чтобы изменить маршрут, откроем и изменим файл routes\web.php (показана только часть файла):

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

Route::get("/contents/create", "MainController@input")**->name("contents.create")**;

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

**Route::get("/cont/{content}/edit", "MainController@input");**

**Route::get("/cont/{id}/delete", "MainController@destroy");**

На данный момент наш MainController выглядит следующим образом:

<?php

namespace App\Http\Controllers;

use App\Content;

use App\Http\Requests\ContentRequest;

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()]);

}

public function input(Content $content) {

return view ("contents.input", ["content" => $content]);

}

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");

}

Подключение фасада DB нам здесь больше не нужно, поэтому я его уже удалил, удалите и вы.

Для создания метода удаления записей, необходимо описать его в контроллере, добавив в MainController следующий метод:

public function destroy(Content $content) {

Content::destroy($content->id);

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

->with("status", "Раздел " . $content->id . " удалён");

}

Мы воспользовались опять моделью Content и методом destroy() класса Model. Метод destroy() позволяет удалять модель (запись) не получая ее, если известен ее первичный ключ – id. Передав id в метод destroy(), мы удаляем запись и делаем перенаправление на действие метода контроллера MainController all(), передав строковое сообщение для секции status о том, какой раздел был удален. Подправим код шаблона resources\views\contents\all.blade.php и приведем его к виду (изменения выделены жирным шрифтом):

@extends('layouts.app')

@section("title", "Все разделы")

@section("main")

<div class="container top60">

<h1>Все разделы</h1>

<p>

**@section("status")**

**@endsection**

@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>

<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]) }}" class="btn btn-primary">

Редактировать

</a>

</p>

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

Удалить

</a>

@endif

</th>

</tr>

@endforeach

</tbody>

</table>

</div>

</div>

@endsection

Это все, что мы хотели сделать.

В заключении мы можем 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.