**КЛОНИРОВАНИЕ С GIT**

* Clone your project
* Go to the folder application using cd command on your cmd or terminal
* Run composer install on your cmd or terminal
* Copy .env.example file to .env on the root folder. You can type copy .env.example .env if using command prompt Windows or cp .env.example .env if using terminal, Ubuntu
* Open your .env file and change the database name (DB\_DATABASE) to whatever you have, username (DB\_USERNAME) and password (DB\_PASSWORD) field correspond to your configuration.
* Run php artisan key:generate
* Run php artisan migrate
* Run php artisan serve
* Go to http://localhost:8000/

**ПОДКЛЮЧЕНИЕ**

-Устанавливаем Composer

-Через командную строку устанавливаем Laravel(как указано на сайте Laravel)

composer create-project laravel/laravel example-app ("8.5.\*"-если надо версию указать)

npm install --save-dev bootstrap@4(установка 4 версии)

https://startbootstrap.com/templates(шаблоны)

-Устанавливаем наш новый проект Laravel в C:\OSPanel\domains(если установлен Open Server)

-В настройках Open Server и в разделе "Домены" настроим новый домен для нашего нового проекта. Домашний каталог у Laravel (там где лежит файл index.php) находится в каталоге "public", и именно его мы должны указать в качестве папки домена. Имя домена "mysite.loc" может быть любое



**СТРУКТУРА И ИМЕНА ФАЙЛОВ ПРИ CRUD**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Verb**(метода запроса в Route) | **URI**(имя страницы в Route) | **Action**(имя функции в Controller) | **Route Name**(имя страницы внутри проекта) | **Пояснение** |
| GET | /photos | index | photos.index | отображение на странице ВСЕХ данных из БД(список всех отзывов о товаре) |
| GET | /photos/create | create | photos.create | страница на которой заполняются данные для внесения в БД(форма в которой пишешь свой отзыв, свои данные)  в Controller:  public function store() { $data=request()->validate([ 'title'=>'string', ]); Post::create($data);  return redirect()->route('post.index'); } |
| POST | /photos | store | photos.store | отправка данных в БД  в View:  <form action=  "{{route('post.store')}}" method="POST">  @csrf-токен надо указывать при использовании метода POST |
| GET | /photos/{photo} | show | photos.show | отображение на странице ОДНОГО элемента из БД(один подробный отзывов о товаре)  в Controller:  public function show(Post $post) { return view('post/show', compact('post')); }  Post-Модель  $post-то что и имени запроса в { } но с $ |
| GET | /photos/{photo}/edit | edit | photos.edit | страница с отображением ОДНОГО элемента из БД, который можно редактировать(редактирование своего отзыва о товаре) |
| PUT/PATCH | /photos/{photo} | update | photos.update | отправка изменений в ОДНОМ элементе в БД  в Controller:  public function update(Post $post) { $data = request()->validate([ 'title' => 'string', ..., ]); $post->update($data); return redirect()->route('post.show', $post->id); }  в View:  <form action="{{route('post.update')}}" method="POST"> @csrf @method('patch') |
| DELETE | /photos/{photo} | destroy | photos.destroy | удаление ОДНОГО элемента из БД  в Controller:  public function destroy(Post $post) { $post->delete(); return redirect()->route('post.index'); }  в View:  <form action="{{route('post.delete', $post->id)}}" method="POST"> @csrf @method('delete') <button type="submit" >Delete</button> </form> |

**ФУНКЦИИ(ХЭЛПЕРЫ)**

**protected $table='имя таблицы';** – в Model указывать имя той таблицы с которой связана Model

**dd(переменная)** – показывает значение и останавливает выполнение кода

**dump(переменная)** – показывает значение без остановки выполнения кода

**ELOQUENT ORM**

Laravel содержит [ORM-библиотеку](https://ru.wikipedia.org/wiki/ORM) Eloquent, которая работает с базой данных. Eloquent использует связь Model со своей БД.

**find(id)** – возвращает элемент из БД соответствующий id

*$user = User::find(1);*

**create([])** –добавляет элементы в БД(в Model необходимо вставить protected $guarded=[] или protected $guarded=false;)

$user = User::create([

‘title’=>’some’,

…]);

**update([])** –изменяет существующий элементов в БД

$user = User::find(…)->update([

‘title’=>’update some’,

…]);

**delete()** –удаляет существующий элементов в БД(В Миграции(создание таблицы) указать **$table->SoftDeletes()**-данные останутся в таблице и их можно восстановить и в Model указать use SoftDeletes;)

$user = User::find(…)->delete();

**restore()** –восстанавливает удаленный элементов в БД(если он был удален при помощи SoftDeletes)

$user = User:: withTrashed() ->where('airline\_id', 1) ->restore()

**firstOrCreate([проверка], [новый элемент])** –возвращает элемент БД соответствующий условиям(например значение определенного атрибута), а если такого нет то создаст Новый и возвращает его. Можно использовать, чтобы избежать дублирования

**updateOrCreate([проверка], [обновление элемента или новый элемент])** – тоже что и firstOrCreate(), только если находится элемент соответствующий условию в него вносятся изменения и возвращает измененный элемент

**МЕТОДЫ LARAVEL**

**all()** –возвращает массив из Коллекция(collect). Позволяет получить все элементы БД

$user = User::all();

**where(‘ключ’, ‘значение’)** –фильтрует Коллекция(collect) по “ключ” / “значение”

$posts=User::where('is\_published', 1)->get();

-or-

$posts=User:: all()->where('is\_published', 1);

**first()** –возвращает первый элемент Коллекция(collect). Позволяет получить первый элемент БД

$user = User::first();

**total()** –возвращает количество элементов массива, в котором элементы из БД

**CONTROLLER**

Создание Controller:

-в командной строке php artisan make:controller [Name(CamelCase)Conroller]

Структура

**MIGRATION**

Создание Migration(создание таблицы) c Model

-в командной строке php artisan make:model [Name] -m

-подключение к базе: файл .env заменить DB\_DATABASE на имя вашей базы

Накатить(выполнить функцию up) Migration

-в командной строке php artisan migrate

Oткатить(выполнить функцию down) Migration

-в командной строке php artisan migrate:rollback

Обновить Migration

-в командной строке php artisan migrate:fresh

Добавить колонку в Migration

-в командной строке php artisan make:migration [add\_colomun\_’имя атрибута’\_to\_’имя миграции где создавалась таблица(2023\_01\_24\_084927\_create\_posts\_table.php)’]

public function up() {

Schema::table('posts', function (Blueprint $table) {

$table->text('description')->nullable();

}); }

public function down() {

Schema::table('posts', function (Blueprint $table) {

$table->dropColumn('description'); }); }

Удалить колонку в Migration

-в командной строке php artisan make:migration [delete\_colomun\_’имя атрибута’\_to\_’имя миграции где создавалась таблица(2023\_01\_24\_084927\_

create\_posts\_table.php)’]

public function up() {

Schema::table('posts', function (Blueprint $table) {

$table->dropColumn('description');

}); }

public function down() {

Schema::table('posts', function (Blueprint $table) {

$table->text('description')->nullable();

}); }

Редактировать колонку(имя,тип данных..) в Migration

Если будет ошибка при php artisan migrate, то выполнить composer require doctrine/dbal

-в командной строке php artisan make:migration [edit\_colomun\_’имя атрибута’\_to\_’имя миграции где создавалась таблица(2023\_01\_24\_084927\_create\_posts\_table.php)’]

public function up() {

Schema::table('posts', function (Blueprint $table) {

!name!

$table->renameColumn('content', ‘post\_content’);

!type!

$table->text('column\_name')->change();

}); }

public function down() {

Schema::table('posts', function (Blueprint $table) {

!name!

$table->renameColumn(' post\_content ', ‘content’);

!type!

$table->string('column\_name')->change();

}); }

Удалить таблицу в Migration

-в командной строке php artisan make:migration [delete\_’имя миграции где создавалась таблица(2023\_01\_24\_084927\_create\_posts\_table.php)’]

public function up() {

Schema::dropIfExists('posts')

public function down() {

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

$table->id();

$table->text('description')->nullable();

…}); }

**VIEW(with Bootstrap)**

В Contoller в функции указываем – return view(‘file in folder view’, compact(таблица)), данные из compact(таблица) передаются в файл в папке view

Подключение Bootstrap

– composer require Laravel/ui -> php artisan ui bootstrap -> npm run dev

<link rel="stylesheet" href="{{asset('css/app.css')}}"> - подключаем на странице стили bootstrap, с помощью asset путь всегда будет прописан правильно, даже если имя файла blade.php поменяется или его расположение

**@foreach**(array as item){{item}}**@endforeach** –перебор массива в файлах blade.php, при этом item выводится на html странице

**@yield(‘name’)** –создается шаблон на html странице с именем ‘name’

**@extends (‘folder/file’)** –на другой странице blade.php вставляется этот шаблон, с указанием пути к файлу

**@section** (‘name’) <div>…</div> **@endsection** – в этой же странице где @extends вставляется @section с тем же именем ‘name’ которое указано в @yield(‘name’)

**->name(‘name’)** –присваиваешь имя страницы в Route и далее в проекте можно использовать это name когда надо указать ссылку на страницу…

<a href="{{route('main.index')}}">Main</a>

**@error**(‘name’/связано с атрибут name=’name’ например в input){{ }} **@enderror** – проверяет введено ли значение, если нет выводит <Тег>

@error('title')

<p class="text-danger">{{$message/ переменная из Laravel выводит какая ошибка }}</p>

@enderror

в Controller

$data = request()->validate/проверяет соответствуют ли введенные данные уловию в Request(например тип данных правильно ди ввежен)([ 'title' => 'required|string',…])

**old('title')-**сохраняет значение при обновлении если оно введено, а в других полях нет

<input type="text" name="title" value="{{ old('title') }}">

**VIEW(with VUE+ElementPlus)**

Подключение VUE + ElementPlus UI

– composer require Laravel/ui -> php artisan ui vue -> npm install element-plus –save -> add in app.js: ->npm run dev

import ElementPlus from 'element-plus'

import 'element-plus/dist/index.css'

app.use(ElementPlus)

- регистрация компонента VUE в файле app.js

import ExampleComponent from './components/ExampleComponent.vue';

app.component('example-component', ExampleComponent);

- использование компонента VUE в файле blade.php

<head>

@vite('resources/js/app.js')

</head>

<body>

<div id="app">

<example-component></example-component>

</div>

</body>

Структура и атрибуты VUE

вложенность одного компонента в другой

<template>

    <div class="container">

        <ExampleComponent></ExampleComponent> - отображение компонента

    </div>

</template>

<script>

import ExampleComponent from './ExampleComponent.vue'; - импортируем нужный компонеyт(без объявления в app.js)

export default {

    components: {

        ExampleComponent - такое же имя как имя файла

    },}

</script>

зарезервированный ключ data(сохранение данных "не изменных")

<template>

    <div class="container">

        <div>{{ name }}</div> - отображение данных

           </div>

</template>

<script>

export default {

       data(){

        return{

            name: 'Jan', - сохранение данных

        }},}

</script>

методы(запуск их по событию)

<template>

    <div class="container">

        <button @click="changeName">Click</button>-выполняем функцию по нажатию

    </div>

</template>

<script>

export default {

    methods: {

        changeName() {

           console.log('Hello') - создаем функцию

        }}}

</script>

вычисляемые свойства (computed)

Частая область их применения — красивое отображение данных для пользователя. Это может быть вывод даты в подробном формате или форматирование текста, выбор одного из варианта по условию

<template>

      <div> <p>{{ title }}</p>

     <p>{{ timeago }}</p> </div>

</template>

<script>

export default {

data(){

        return{

                name: String,

                birtday: String

}},

computed: {

        title: function() { return 'Пользователь: ' + this.name.toUpperCase(); },

        timeago: function() { return moment(this.birthday).fromNow(); }

}}

</script>

директива v-for(Для работы с массивами)

<ul>

        <li v-for="user in users">{{ user}}</li>

</ul>

<script>

data(){

        return{

users: ['Tom', 'Sam', 'Alice', 'Kate']

}},

</script>

директива v-if(условие)

<div v-if="type === 'A'"> A </div>

Axios c Vuejs и Laravel(RouteServerProvider раскомментирвать 29 строку, чтобы заработал роутинг)

-get

<script>

export default {

data(){

        return{

                posts: null,

}},

mounted()-метод жизненного цикла запускает выполнение getPosts{

     this.getPosts()

}

    methods: {

        getPosts() {

          axios.get(url: '/posts').then(item =>-данные с контроллера(бэк) приходят в виде объекта с ключем data а значение это наши данные с бэка {

                 this.posts = item.data -записали данные в posts(метода data) и только теперь можно их получить для фронтэнда

              }.catch(error=>{}).finally({})

           )

        }}}

</script>

-post

<template>

<div>

<input v-model="name"-введенное значени попадает в data()->name id="name" type="text" />

<button type="button" @click.prevent="addClient"></button>

</div>

</template>

<script>

export default {

data() { return { name: null } },

methods: {

addClient() {

axios .post('api/client', { name -имя такое как колонка в базе: this.name }) .then(res => { })

} }, }

</script>

**ОДИН КО МНОГИМ**

в Migration

$table->unsignedBigInteger('category\_id)->nullable();-создает колонку с которая связана с другой таблицей, правильно использовать имя Модели с \_id

$table->index('category\_id', 'post\_category\_idx');-это колонке присваиваем Индекс

$table->foreign/Внешний ключ('category\_id'/Имя колонки которая связана с другой таблицей', 'post\_category\_fk')->on/С какой таблицей связь('categories')->references('id')/Колонка в другой таблице с которой связь;-устанавливаем связь между двумя таблицами

Быстрый доступ к связанным элементам из БД

в Controller

public function index() {

$categories=Category::find(1);

$categories->posts/имя функции из модели, без () в конце; }

в Model

class Category extends Model {

public function posts() {

1 вариант - return $this->hasMany(Post::class/Другая модель в связке, 'category\_id'/имя колонки с внешним ключом, 'id'/имя колонки из другой таблицы с которой связь); } }

(return $this-> belongsTo(...)-одно значение ищем, выше много)

2 вариант-Имя колонки которая ссылается на другую таблицу должно быть как и имя Модели\_id для 2 таблицы и:

return $this->hasMany(Post::class), остальные атрибуты не нужны

**МНОГИЕ КО МНОГИМ**

в Migration(есть 1 и 2 таблицы, чтобы сделать связь между ними многие ко многим создается таблица 3, в 3 уже создаётся две связи с один ко многим с 1 и 2 таблицей)

php artisan make:migration create\_post\_tag\_table/Имена 3 таблицы должно быть как имя 1 Модели и 2 Модели к которым привязаны таблицы

$table->unsignedBigInteger('tag\_id');

$table->unsignedBigInteger('post\_id');

$table->index('post\_id', 'post\_tag\_post\_idx');

$table->index('tag\_id', 'post\_tag\_tag\_idx');

(связь с 1 таблицей)$table->foreign('post\_id'/Имя колонки которая связана с другой таблицей', 'post\_tag\_post\_fk')->on('posts'/С какой таблицей связь)->references('id') )/Колонка с которой связь;

(связь с 2 таблицей)$table->foreign('tag\_id', 'post\_tag\_tag\_fk')->on('tags')->references('id');

Быстрый доступ к связанным элементам из БД

в Controller

public function index() {

$tag=Category::find(1);

$tag->posts/имя функции из модели,без () в конце; }

в Model

class Tag extends Model {

public function posts() {

1 вариант - return $this-> belongsToMany (Post::class/Другая модель в связке, ‘post\_tags’/Таблица 3 связуюшая 1 и 2, 'tag\_id'/имя колонки с внешним ключом в модели Tag, 'post\_id'/ имя колонки с внешним ключом в модели Post); } }

2 вариант -при создании миграции имя 3 таблицы должно быть php artisan

make:migration create\_1таблица\_2таблица\_table (Таблица 1 и 2 в ед.числе). Имя колонок в 3 Таблице, должны быть как имя Модели1\_id и Модели2\_id и:

return $this-> belongsToMany (Post::class), остальные атрибуты не нужны

-для того чтобы принимал в тегах ‘name’ большего 1 значения необходимо поставить [] name=’tags[]’

-из одной формы передать данные в 1 и 2 таблицы

В Controller

$tagss = $data['tags']; - данные для 2 таблицу

unset($data['tagss']); - удаляем из общего массива данные для 2 таблицы

$post = Post::create($data);

$post ->tags()/функция из Модели Post->attach ($tagss);/Метод Laravel- записывает в 3 Таблицу все taggs которые принадлежат одному $post

$post ->tags()/функция из Модели Post->sync() ;/ Метод Laravel-редактирует в 3 Таблице taggs принадлежат одному $post

$post ->tags()/функция из Модели Post->detach() ;/ Метод Laravel-удаляет из 3 Таблицы taggs принадлежат одному $post

В attach,sync,detach можно указывать как 1 элемент, так и массив.

В 3 таблицы можно использовать дополнительные зависимости с помощью Pivot:

в Model

return $this-> belongsToMany (Post::class)-> withPivot('count') /имя колонки в 3 Таблице;

В Controller

$post ->tags()->attach(1,['count'=>6]); /Записываем в 3 таблицу post привязываем к нему tags и count для этого вторым аргуметом в attach указывается имя колонки и значение

foreach ($post->tags as $tag) { echo $tag->pivot->count; }/получить значения колонки count

**ОДНОМЕТОДНЫЕ КОНТРОЛЛЕРЫ**

в Controller

public function \_\_invoke() { без имени функции }

-если надо однометодные контроллеры сгруппировать в папку то:

Route::group(['namespace'=>'Post/имя папки'], function() {

Route::get('/posts', 'PostController /указывается только имя контроллера без функции ')->name('post.index');

…})

namespace App\Http\Controllers\Post-указать путь к папке к сгруппированным контроллером, стоит по умолчанию namespace App\Http\Controllers;

**КЛАСС REQUEST(ДАННЫЕ ИЗ ФОРМЫ)**

Для получения данных формы в Laravel используется специальный объект запроса Request. Создание:

php artisan make:request [ИмяRequest] или если однометодные контроллеры php artisan make:request Post(имя папки)\[ИмяRequest]

в созданном файле:

public function authorize() {

return true; }

public function rules() {

return [

'title' => 'required|string'

…]; }

в Controller

public function \_\_invoke(StoreRequest $request) {

$data = $request->validated();

…}

**КЛАСС SERVICE(ЛОГИКА ПО CRUD БД)**

Создается для вынесения основной логики по CRUD(например-внесение данные из формы в БД) в отдельный класс, который взаимодействует со своим Controller. Что позволяет лучше понимать код

-в папке с однометодными контроллерами создаем BaseController.php в нем

class BaseController extends Controller {

public $service;

public function \_\_construct(Service/имя класса в файле Service

$service/задаем переменную-экземпляр класса) {

$this->service = $service; } }

-в тех однометодных контроллерах где подключаем Service надо наследование поменять на BaseController

class UpdateController extends BaseController {

public function \_\_invoke(Post $post, UpdateRequest $request) {

$data = $request->validated();

$this->service->update($post, $data/данные которые надо передать в Service); } }

-в папке app создаем Service там папку такую же как у однометодных контроллеров в ней файл Service

class Service {

public function update($post, $data) {

логика по CRUD} }

**КЛАССЫ FACTORY И SEED(ТЕСТОВЫЕ ДАННЫЕ В БД)**

Laravel имеет простой метод заполнения базы данных тестовыми данными, используя классы-наполнители(seed classes). Можно данные внести вручную, но если нужен

большой объем данных в БД можно в Seed использовать автоматический метод создания

Factory

php artisan make:factory /Имя/Factory -m/Имя модели с которой связываем/

в Factory

public function definition() { return

[ 'title' => 'string',/указываем колонку и значение

'post\_content' => $this->faker/метод Laravel->text/подставит рандомный текст

'is\_published' => Category::get()->random()->id,

]; }

в Seeders

php artisan make:seeder /Имя/Seeder

public function run() {

Post/Имя модели с которой связь::factory(10/количество записей в таблицу)->create();}

в DatabaseSeeder

указываем ссылку на наш созданные Seeder

public function run() {

$this->call(PostSeeder::class); }

php artisan migrate –-seed – запускаем выполнение Seeder

**ПАГИНАЦИЯ**

Пагинация на сайте(для увеличения производительности) — это разбиение контента сайта по отдельным страницам. Для перехода пользователей по этим страницам используется навигационный блок — пагинатор(например – список товаров в интернет магазине разбит на несколько страниц)

в Controller

public function \_\_invoke() {

$posts = Post::paginate/ метод Laravel (5/ указываем количество записей);

return view('post/index', compact('posts')); }

подключить Bootstrap

php artisan vendor:publish --tag=laravel-pagination

в app\Providers\AppServiceProvider.php добавить:

public function boot() {

Paginator::defaultView('vendor\pagination\bootstrap-4'/путь в файлу bootstrap-4); }

в View

<div> {{$posts->links()/ метод Laravel }} </div>

**FILTER, ФИЛЬТРАЦИЯ ДАННЫХ**

В Request создаем файл Filter в котором указываем ключи по которым будет производится фильтрация.

public function rules() {

return [ 'title' => 'required|string',

'post\_content' => 'string',

...]; }

При выполнение метода Get в строке запроса (Query string) передаются данные([http://mysite.loc/posts?title=](http://mysite.loc/posts?page=1)Mr) Для получение входных данных из строки запроса

в Controller

public function \_\_invoke(FilterRequest $request) {

$data=$request->validate();/получаем данные по методу Get через строку запроса

$query=Post::query();/вылавливает передаваемые данные в строку запроса по методу Get для фильтрации

if (isset($data['title'])/isset проверяет есть ли/введено значение) {

$query->where('title', 'like', "%{$data['title']}%"); /если значение есть, то фильтрует колонку ‘title’ оставляя только значение равные $data['title'], %-обозначает поиск в любом месте слова); }

if (isset($data['category\_id'])) {

$query->where('category\_id', "%{$data['category\_id']}%"); }

$posts = $query->paginate(10); /если фильтрация сработала выводим по 10 записей, если не было фильтрации выводим все записи по 10

return view('post/index', compact('posts')); }

в View

<div> {{$posts->withQueryString()->links()// два метода Laravel }} </div>/

withQueryString() при пагинации, переход на новую страницу не пропадает фильтрация

**AdminLTE**

<https://adminlte.io/> -> DownLoad -> Source code(zip) -> код из index.html скопировать в свой файд blade.php -> папки распаковать plugins и dist в проект в public

**АВТОРИЗАЦИЯ В LARAVEL**

Готовая форма для авторизации Пользователей средствами Laravel php artisan ui:auth. В web.php появится и создадутся все необходимые файлы в том числе в БД появится таблица users

Auth::routes();

Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');

Теперь можно перейти '/home’ и увидеть панель регистрации

**КОРЗИНА ПОКУПОК**

Готовая библиотека для добавления, редактирования, удаления продуктов в корзину, получения их количества, подсчет цены...Работа построена на Session/сессии

composer require bumbummen99/shoppingcart

php artisan vendor:publish --provider="Gloudemans\Shoppingcart\ShoppingcartServiceProvider" --tag="config"

Cart::instance('default'/имя экземпляра - если надо поместить товары в корзину и в избранное два имени указывается/)->add/добавление товара в сессию, ключи(колонки таблицы) изменить нельзя, в options могут быть любые ключи/ ([

'id' => ' ',

'name' => ' ',

'qty' => ' ',

'price' => ' ',

'weight' => 0,

'options' => [ '' => , '' => , ... ] ])-associate('App\Models\Product');

Cart::instance('default')->content()->all(),/получить все добавленнеы товары/

Cart::instance('default')->total(),/получить стоимость товаров в корзине, цена 1 товара это qty\*price/

Cart::instance('default')->count(),/получить общее количество товаров/

Cart::instance('default')->update(rowId, quantity); /изменить количество товара qty/

Cart::instance('default')->remove($id); /удалить товар из сессии/

**MIDDLEWARE**

Мидлвар обеспечивает удобный механизм для проверки и фильтрации HTTP-запросов, поступающих в Ваше приложение. Мидлвар является посредником между HTTP-запросом и Controller, если проверка в мидлвар выполняется тогда выполняется действие в Controller. Например, Laravel включает мидлвар, который проверяет, аутентифицирован ли пользователь Вашего приложения. Если пользователь не аутентифицирован, мидлвар перенаправит пользователя на экран входа в систему Вашего приложения. Однако, если пользователь аутентифицирован, мидлвар позволит запросу продвинуться дальше в приложение.

**@auth( )**-возвращает данные из БД о пользователе который сейчас находится на сайте

создание:

php artisan make:middleware AdminPanelMiddleWare

Регистрация(app\Http\Kernel.php):

'admin'/задаем имя => AdminPanelMiddleWare::class,

в Controller

Route::group(['namespace' => 'Admin', 'prefix' => 'admin', 'middleware' => 'admin'], function () {...} – указываем имя мидлвар и данный Controller будет выполнять проверку

в AdminPanelMiddleWare

public function handle(Request $request, Closure $next) {

if (auth()->user()->role !== 'admin') { return redirect('home'); }/указываем условия когда выпонится Controller

return $next($request); }

**POLICY & GATE**

Политики(Policy) — это специальные классы, которые предоставляют методы авторизации(разрешения) действия пользователя. Тогда как Шлюзы(Gate) — противоположный по назначению механизм. Его задача — запрещать все действия, которые не разрешены политикой. Таким образом шлюзу необходимо указать, какого типа ресурс руководствуется какой политикой

создание:

php artisan make:policy PostPolicy --model=Post

в app\Policies\PostPolicy.php

public function view(User $user, User $model/в переменной хранится данные о пользователе) {

return $model->role !== 'admin';} /условие

регистрация(app\Providers\AuthServiceProvider.php):

protected $policies = [

User::class=>PostPolicy::class, ];

использование(в Controller):

this->authorize('view', auth()->user()) /красным тоже что и в создании

использование(в View):

@can('view', auth()->user()) /красным тоже что и в создании

<a class="nav-item nav-link" href="{{route('admin.post.index')}}">Admin</a> @endcan

**EMAIL**

В Laravel есть встроенная библиотека guzzle отвечающая за отправку писем

создание:

php artisan make:mail OrderIn – письмо внутреннее

php artisan make:mail OrderOut – письмо для заказчикаа

во view созжаем два файля order\_out и order\_in

в OrderIn и OrderOut

class OrderIn extends Mailable {

use Queueable, SerializesModels;

public $orderOut;

/\*\* \* Create a new message instance. \*

\* @return void \*/

public function \_\_construct($orderOut) {

$this->orderIn = $orderout; }

/\*\* \* Build the message. \*

\* @return $this \*/

public function build() {

return $this->view('mail.order\_out', $this->orderOut); } }

отправления письма после сохранения заказа в Controller

Mail::to($request->user()/адрес почты заказчика)->send(new OrderOut([...]));

Mail::to(degree\_183@mail.ru/адрес своей почты)->send(new OrderIn([

'session\_products' => Cart::instance('default')->content()->all(),

'session\_total' => Cart::instance('default')->total(),])/данные для view order\_out);

в env.

MAIL\_MAILER=smtp

MAIL\_HOST=smtp.gmail.com

MAIL\_PORT=465

[MAIL\_USERNAME=teplohouses@gmail.com](mailto:MAIL_USERNAME=teplohouses@gmail.com)

MAIL\_PASSWORD= / гугл аккаунт-безопасность-пароли приложений-создать

MAIL\_ENCRYPTION=SSL

[MAIL\_FROM\_ADDRESS=teplohouses@gmail.com](mailto:MAIL_FROM_ADDRESS=teplohouses@gmail.com)

MAIL\_FROM\_NAME="SHOP"

order\_in.blade.php и order\_out.blade.php HTML страницы, верстка из которых будут приходить на почту

**API**

добавить новый адрес api

-папка routes->файл api.php

Route::resources([ 'index' => ApiController::class,

'show' => ApiController::class, ]); - index, show.. - методы в Controller и одновременно имя в url (/api/index)

создать Controller для api

-php artisan make controller: Api(новая папка в папке Controller)/ApiController --api

в ApiController функции

index - возвращение всего списка

public function index() {

return Category::all();

}

store - сохранить элемент

show - возвращение одного элемента

public function show(id) {

return Category::find(id);

}

update - изменить элемент

destroy - удалить элемент

Ресурсы (Resources) — дополнительный слой поверх Eloquent-моделей. Они позволяют поддерживать единый формат возвращаемых данных. Плюс позволяют отсекать лишние или секретные данные, например, пароли и токены.

-php artisan make controller: resource ApiResource

в ApiResource.php:

public function toArray($request) {

return[

/указываем поля из базы которые возвращаются/

‘id’=>$this->id;

‘name’=>$this->name;];}

в ApiController.php:

public function index() {

return ApiResource::collection(Category::all());

}

связь один ко многим:

-модель всё так же

-в ApiController.php:

public function show(id) {

return new ApiResource/при возврате 1-го объекта коллекцию не используется/(Category::with(‘posts’/ф-я один ко многим из модели)->find(id));

}

- в ApiResource.php (включить в массив):

‘posts’->CategoryPostResource/cоздаем новый Resource и указываем какие поля нам надо возвращать::collection($this->posts);

добавление своих данных при возврате данных запрошенных:

- в ApiResource.php (включить в массив):

‘item’=>$this->collection;/все данные при возврате попадают в массив item: [{‘id’:…, ‘name’:…}] и ниже содаем другой массив meta с произвольными данными которые так же вернутся

‘meta’=>[

‘param1’=>’1’

‘param2’=>’2’]

добавление новых данных:

-в ApiController.php:

public function store(PostStoreRequest $request) {

new\_post=Post::create($request->validated());

return new ApiResource(new\_post);

}

изменение данных:

-в ApiController.php:

public function update(PostStoreRequest $request, Post $post) {

$post-> update ($request->validated());

return new ApiResource($post);

}

в PostMan метод Post, а в теле передаем указать \_method-PUT

удаление данных:

-в ApiController.php:

public function destroy(Post $post) {

$post-> delete();

return response(null, Response: HTTP\_NO\_CONTENT);

}

в PostMan метод Post, а в теле передаем указать \_method-DELETE