База знаний по Laravel

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

\*\*\* Вспомогательные инструменты Ларавель:

dd() - выводит данные и останавливает скрипт

dump() - то же, только без остановки

Коллекции - альтернатива массивам в Ларавель

asset() - генерирует путь (преобразуя в абсолютный) к ресурсам в папке public

\*\*\*

\*\*\* Миграции:

php artisan make:model /\* Название модели \*/ -m (создание модели и миграции для соответствующей таблицы)

php artisan migrate - накат всех миграций

php artisan migrate:rollback - откат всех миграций

При откате сначала откатываются миграции, созданные позднее

При создании модели вместе с миграцией Ларавель создаёт таблицу с названием во мн. числе

Для создания возможности sortDeletes в миграции создания таблицы прописываем $table->softDeletes();

Обновляем миграций: php artisan migrate:fresh

Пример создания новой миграции: php artisan make:migration add\_column\_description\_to\_posts\_table (to\_posts\_table - ключевые слова для привязки миграции к нужной таблице)

Проверка корректности созданной миграции: накат->откат->накат

Для добавления колонок:

1. В up: $table->тип('название')->nullable()->after('название колонки, после которой надо поставить новую'); // нужно ставить nullable() для корректной работы

2. В down: $table->dropColumn('название');

3. Выполнить накат миграций (php artisan migrate)

Для переименования колонки:

1. Установть библиотеку doctrine/DBAL: composer require doctrine/dbal

2. В up: $table->renameColumn('текущее\_название', 'новое\_название');

3. В down: $table->renameColumn('новое\_название', 'текущее\_название');

4. Выполнить накат миграций (php artisan migrate)

Для изменения типа поля:

0. Установть библиотеку doctrine/DBAL: composer require doctrine/dbal

1. В up: $table->новый\_тип('название\_поля')->change();

2. В down: $table->текущий\_тип('название\_поля')->change();

Создание связей в таблицах (пример):

1. $table->unsignedBigInteger(‘category\_id’)->nullable(); // создаём колонку внешнего ключа (связанная таблица - categories);

Не обязательно ( $table->index(‘category\_id’, ‘post\_category\_idx’) // создаём индекс)

2. $table->foreign(‘category\_id’)->on(‘categories’)->references(‘id’)->onDelete(‘cascade’); // foreign – указание, что внешний ключ, on – на какую таблицу ссылается, references – название поля ключа в связанной таблице,

‘cascade’ – каскадное удаление (при удалении записи удалятся все связанные с ней внешними ключами записи)

\*\*\*

\*\*\* Роутинг:

Для использования записи в роутах вида 'ControllerName@methodName' в app/Providers/RouteServiceProvider раскомментируем строку "protected $namespace = 'App\\Http\\Controllers';"

Если название роута состоит из нескольких слов, лучше разделять слова нижним подчёркиванием (например 'first\_or\_create')

Для правильного формирования ссылок (в href ссылок страницы) следует использовать именование роутов, к примеру: Route::get('posts', 'PostController@index')->name('post.index');

Пример записи ссылки на страницу: <li><a href=" {{ route('post.index') }}">Статьи</a></li>

Конструкция перевода пользователя на определённый url:

return redirect()->route('post.index'); // где post.index – имя роута, заданное в “name” web.php

Пример роута на страницу с параметром: Route::get(‘posts/{post}’, ‘PostController@show’)->name(‘posts.show’);

Пример роута на страницу редактирования: Route::get(‘posts/{post}/edit’, ‘PostController@edit’)->name(‘posts.edit’);

Группировка роутов:

Route::group([‘namespace’ => ‘// название папки в контроллерах’], function(){

/\* Роуты, ведущие на контроллеры, относящиеся к одной сущности (например Post). При этом, методы контроллеров убираются, остаётся только название контроллера \*/

});

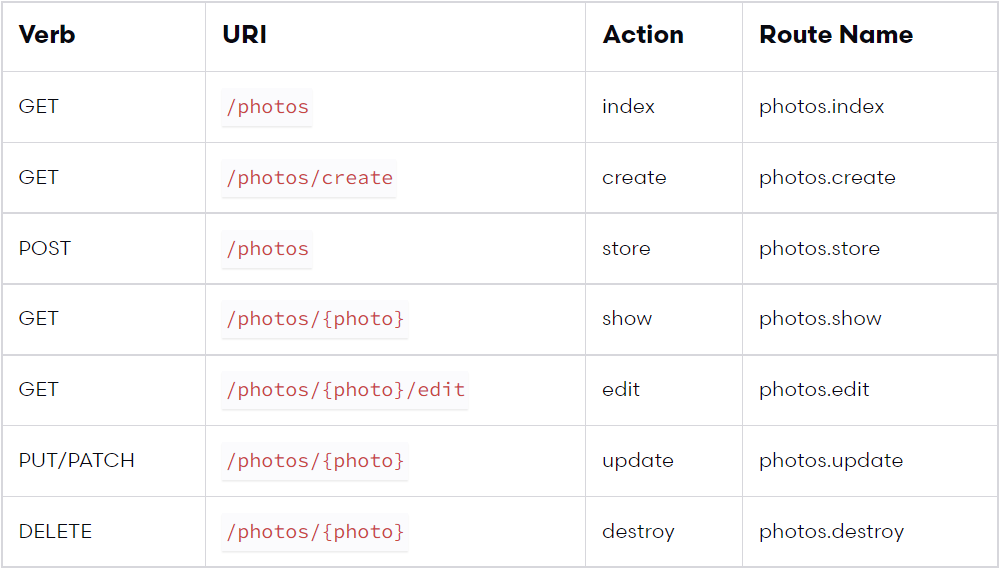
\*\*\*

\*\*\* Контроллеры:

Можно копировать контроллеры

Контроллеры именуются в единственном числе, например PostController

CRUD нотации:



Пример организации валидации форм в контроллере:

$data = request()->validate([

            'title' => 'string',

            'content' => 'string',

            'image' => 'string'

        ]);

// Однометодные контроллеры

Размещаются в Controllers в отдельных папках по логике (например. контроллеры постов в папке Post)

Метод таких контроллеров: \_\_invoke (public function \_\_invoke(//аргументы){})

\*\*\*

\*\*\* Модели:

Модели именуются в единственном числе с большой буквы, например: Post

Для явного указания того, с какой таблицей связана модель, используется свойство: protected $table = 'название таблицы'; (в практике следует указывать явно!)

Для того, чтобы дать разрешение на запись/изменение полей в таблице, в модели указываем protected $guarded = []; (или false), либо там же создаём protected массив fillable и указываем список атрибутов, разрешённых к работе

Получение данных из БД:

Пример операции получения данных по id: $post = Post::find(1); (метод find с передачей id)

Пример операции получения всех данных: $posts = Post::all(); (метод all)

Пример операции выборки по условию: Post::where('is\_published', 1)->get(); (возвращается коллекция) ->first() - вернёт первый найденный элемент

Чтобы находить записи в т.ч. в корзине: $post = Post::withTrashed()->find(3);

Для восстановления записи: $post->restore();

Добавление данных в БД:

Post::create($item); // если добавляем целый массив, Post::create(['field1' => $value1, 'field2' => $value2]); // если добавляем поля по отдельности

Редактирование данных в БД:

1. $post = Post::find(3); // находим редактируемый объект

2. $post->update(['title' => 'updated111', 'content' => 'updated111', /\*'image' => 'updated', 'likes' => 10, 'is\_published' => 1\*/']); (Можно обновлять только нужные поля или все)

Удаление данных из БД (обычный подход):

1. $post = Post::find(3); // находим удаляемый объект

2. $post->delete();

Удаление данных из БД (softDeletes):

1. В миграции создания таблицы прописываем $table->softDeletes();

2. В модели добавляем трейт: use SoftDeletes; и подключаем его в файл: use Illuminate\Database\Eloquent\SoftDeletes;

3. Обновляем миграции php artisan migrate:fresh

4. Производим обычное удаление

Восстановление записи, удалённой через softDeletes:

1. $post = Post::withTrashed()->find(3);

2. $post->restore();

Комбинированные методы создания и обновления данных:

1. firstOrCreate([// массив полей для проверки], [// массив данных для добавления]) - проверяет, есть в БД поля со значениями из 1 массива. Если есть - возвращает запись, если нет - создаёт запись со значениями полей 2 массива

Пример: $post = Post::firstOrCreate(['title' => 'value'], ['title' => 'new title', 'content' => 'new content' // и тд]); // если в БД есть запись со значением value в поле title - в $post вернётся эта запись, еесли нет, то создастся запись со значениями из 2 массива

2. updateOrCreate() - аналогично с п.1, только редактирование

3. findOrFail – если запись с запрашиваемым id не существует, вернётся ошибка 404. Пример: $post = Post::findOrFail(125);

Если в метод поиска одной записи по id в параметры контроллера передать объявление объекта модели, findOrFail будет выполняться автоматически. Пример: public function show(Post $post){ … // метод контроллера

Route::get(‘posts/post’, ‘имя\_и\_метод\_контроллера’); // web.php

Важно! Имя параметра в роуте должно соответствовать имени обхекта модели (‘/post’ - $post)

Для получения записей одной таблицы в соответствии с индексами другой нужно создать в модели первой таблицы следующий метод (пример):

public function posts(){

        return $this->hasMany(Post::class, 'category\_id', 'id');

    }

В контроллере:

$category = Category::find(1);

 dd($category->posts);

// Post::class – класс модели, связанной с 1 таблицей, ‘category\_id’ – внешний ключ, id – идентификатор ключа во 2 таблице

Для получения записи, к которой относится текущая (например категория поста) необходимо использовать следующий метод (пример):

public function category(){

        return $this->belongsTo(Category::class, 'category\_id', 'id');

    }

// Category::class – класс модели, связанной со 2 таблицей, ‘category\_id’ – внешний ключ, id – идентификатор ключа

В контроллере:

$post = Post::find(1);

        dd($post->category);

Для реализации отношений «многие ко многим» между сущностями используются связующие модели, название – сконкатенированные названия моделей по алфавиту, например: PostTag

Пример миграции для модели связанных таблиц:

$table->id();

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

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

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

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

            $table->foreign('post\_id', 'post\_tag\_post\_fk')->on('posts')->references('id');

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

            $table->timestamps();

// Получение записей, относящихся к выбранной в отношениях «многие ко многим» описано в laravelProject (Модель: PostTags, модель: Post (метод tags(), а также модель Tag (метод posts() ))), в контроллере:

$post = Post::find(1);

dd($post->tags);

или

$tag = Tag::find(1);

dd($tags->post);

Дано: таблицы posts, tags, таблица связей post\_tags (pivot). Пример создания поста и записи в pivot таблице (поста и связанных с ним тегов)

$tags = $data['tags'];

$post = Post::create($data);

        /\*foreach($tags as $tag){

            PostTag::firstOrCreate([

                'post\_id' => $post->id,

                'tag\_id' => $tag

            ]);

        }\*/

        $post->tags()->attach($tags);

// tags – массив тегов из формы (checkbox или multiple select)

\*\*\*

\*\*\* Базы данных (общее):

Отличие Eloquent ORM от Database в том, что в Eloquent существует привязка к модели (и связанной с ней таблицы), а в Database QueryBuilder необходимо явно указывать имя таблицы

\*\*\*

\*\*\* Отображения (view):

Пример отображения страницы из контроллера: return view('posts', compact('posts')); // 1 аргумент - название шаблона в resources/views, 2 аргумент - название переменной, которая передаётся на шаблон из контроллера

Пример отображения данных на шаблоне (blade):

@foreach($posts as $post)

<div> {{ $post->title }} </div>

@endforeach

Примерная структура отображений: resources/views/layouts/общие\_шаблоны

Примерная запись секций (встраиваемых шаблонов):

@extends('layouts.main')

@section('content') // название секции

// содержимое

@endsection

В родительском шаблоне:

// после разметки

@yield('content')

По нотации необходимо создавать папки для хранения шаблонов, относящихся к одной сущности, шаблон отображения сущностей называть index (пример структуры: resources/views/posts/index.blade.php)

Для формирования ссылки в шаблоне используется примерно следующая конструкция: <div><a href=”{{ route(‘posts.show’, $post->id) }}” >Ссылка </a></div> // posts.show – имя роута (name в routes)

Для указания метода формы, которых нет в методах форм (например patch), указывается так: @method(‘patch’) // после тега form, например после @csrf

При формировании ссылки на удаление для CRUD нужно оборачивать ссылку в тег form и ставить @csrf

@error … @enderror – блок для вывода ошибок в формах

Пример: @error

<p class="text-danger">{{ $message }}</p> // message – зарезервированная переменная для вывода ошибок

@enderror

Для того, чтобы сохранить введенные валидные значения используется конструкция: value="{{ old('title') }}" //title - название поля (name)

\*\*\*

\*\*\* Для фронтенда:

Подключение user interface к проекту: composer require laravel/ui

Подключение bootstrap к проекту: php artisan ui bootstrap

Подключение npm: npm install

Перекомпиляция css и js: npm run dev // создаёт директории для css и js в public, обращение следует не к resources, а к public (в целях безопасности)

Для подключения bootstrap на страницу: <link rel="stylesheet" href=" {{ asset('css/app.css') }} ">

asset() - генерирует путь (преобразуя в абсолютный) к ресурсам в папке public

Для создания множественного выбора (например checkbox или multiple select) в name надо указать []

\*\*\*

\*\*\* Отношения (Relationships)

// Отношения «один к одному» (One-To-One)

hasOne(Phone::class) – метод, который используется в модели сущности, одному объекту которой соответствует один объект другой (связанной) сущности (например одному пользователю соответствует один телефон). Вторым параметром можно передать внешний ключ (user\_id)

Пример метода в модели (User):

public function phone(){

return $this->hasOne…

}

Использование: $phone = User::find($id)->phone;

belongsTo(User::class) – метод, который используется в модели одной сущности для связывания её с другой (в примере – с помощью belongsTo можно получить запись пользователя, которому соответствует определённый номер телефона)

Вторым параметром можно передать внешний ключ (user\_id)

Пример метода в модели (Phone):

public function user(){

return $this->belongsTo…

}

Использование: $user = Phone::find($id)->user

// Отношения «один ко многим» (One-To-Many)

hasMany(Post::class, ‘category\_id’, ‘id’) – метод, который используется в модели сущности, одному объекту которой могут соответствовать много объектов другой (связанной) сущности (например только одной категории могут соответствовать много постов).

Пример метода в модели (Category):

public function posts(){

return $this->hasMany…

}

Использование: $posts = Category::find($id)->posts;

Можно добавлять доп функции к posts (where, first и тд), пример: $posts = Category::find($i)->posts()->select(…)->where(..)->first(…);

belongsTo(Category::class, ‘category\_id’, ‘id’) – метод, который используется в модели сущности, несколько объектов которой могут соответствовать одному объекту другой (связанной) сущности (например много постов могут соответствовать одной категории).

Пример метода в модели (Post):

public function category(){

return $this->belongsTo(Category::class, ‘category\_id’, ‘id’);

}

// Отношения «многие ко многим» (Many-To-Many)

Для формирования связей создаётся отдельная таблица. Именование – запись моделей в алфавитном порядке, например: PostTag (таблица post\_tags)

В таблице – id первой и второй связываемой таблицы (post\_id, tag\_id)

Пример: даны таблицы постов и тегов.

В Post:

public function tags(){

return $this->belongsToMany(/\*Класс связанной модели\*/, ‘/\*Название таблицы связей\*/’, ‘post\_id(внешний ключ таблицы текущей модели)’, ‘tag\_id(внешний ключ таблицы связанной модели)’);

}

Использование:

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

$post->tags;

Для второй связанной модели (Tag):

public function posts(){

return $this->belongsToMany(Post::class, ‘post\_tags’, ‘tag\_id’, ‘post\_id’);

}

Использование:

$tag = Tag::find($id);

$tag->posts;

Методы:

attach() – записывает в pivot таблицу массив id записей связанной таблицы (для создания записей).

Пример:

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

$post->tags()->attach($tags);

sync() – удаляет предыдущие связи и записывает новые (для редактирования).

Пример:

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

$post->update($data);

$post->tags()->sync($tags);

\*\*\*

\*\*\* Валидация (Requests)

Для создания класса Request: php artisan make:request NameRequest (если однометодные контроллеры, то прописываем namespace, например – php artisan make:request Post/StoreRequest)

В authorize прописываем true !

Для использования Request: в метод контроллера добавляем соответствующий request с классом (например StoreRequest $request)

Обработка данных в контроллере: $data = $request->validated();

(Данные формы попадают сначала в request, где происходит валидация, затем передаются в контроллер).

В методе rules устанавливаются правила валидации полей, пример:

return [

            'title' => 'required|string',

            'content' => 'string',

            'image' => 'string',

            'category\_id' => '',

            'tags' => ''

        ];