

# Laravel

Данное руководство представляет собой перечень основных особенностей и тонкостей Laravel. Пособие является моим личным, учебником и НЕ ЯВЛЯЕТСЯ профессиональной документацией. В документе могут присутствовать ошибки и неточности.

## Оглавление

Введение.....	2
Laravel.....	3
Composer.....	3
Этапы создания проекта.....	3
Model-View-Controller .....	3
Этапы: .....	4
Http строка .....	5
Request(route).....	5
Controller.....	6
Database (миграции) .....	7
Model.....	11
Модель чтения данных из базы.....	13
Строение Model.....	16
ELOQUENT.....	17
DATABASE .....	17
Методы чтения данных (retrieve) .....	18
Создание объекта в бд (добавление).....	21
Методы обновления данных(update) .....	22
Метод удаления данных(delete) и soft delete .....	23
Комбинированные методы создания и обновления данных.....	25
Миграции. Редактирование миграций .....	27
Создание колонок .....	28
Удаление колонок.....	30
Редактирование колонок .....	30
УДАЛЕНИЕ ТАБЛИЦЫ .....	31
View .....	32
Знакомство.....	32
Шаблоны view.....	33
Bootstrap в laravel.....	37
CRUD через интерфейс .....	38
1. INDEX .....	39
2. CREATE .....	39

3. STORE .....	40
4. SHOW .....	42
5. EDIT .....	43
Update .....	43
6. DELETE.....	44
Отношения один ко многим .....	44
Модификация crud - категории(один ко многим) .....	47
Создание:.....	47
Редактирование .....	48
Отношения многие ко многим .....	49
Модификация crud - Теги(многие ко многим) .....	51
Создание .....	52
Редактирование .....	54
Модификация crud - обработчик ошибок .....	55
Отношения один ко многим и многие ко многим через конвенцию Laravel .	57
Однометодные контроллеры .....	59
Класс Request.....	61
Класс Service .....	63
Классы Factory и Seed .....	67
Класс seed .....	68
Factory (фабрика) .....	69
Совместная работа.....	69
Пагинация в Laravel.....	74
Шаблон Filter, фильтрация данных в Laravel .....	78
НЕИСПОЛЬЗУЕМЫЙ ШАБЛОН .....	78
Используемый шаблон.....	81
Admin LTE в Laravel, устанавливаем админку.....	84
Авторизация Laravel.....	85
Класс Middleware в Laravel (Роли авторизации) .....	87
Класс Policy .....	89
Асинхронный CRUD в Laravel. Приложение Postman. ....	92
Класс Resource в Laravel, асинхронный ответ с бека. Restful API.....	95
Как работает RESTful API? .....	97
HTTP-аутентификация .....	98
OAuth .....	98
JWT Token. Ассинхронные роуты.....	106
CRUD с транзакцией .....	110

## Введение

[Laravel](#) - фреймворк. Система, которая облегчит разработку. Это уже разработанная архитектура программы с готовыми решениями многих проблем. Инструмент для эффективной разработки.

[Composer](#) - пакетный менеджер для php. Нужен для подключения и обновления пакетов и плагинов. Это менеджер зависимостей. (библиотеки, улучшающие разработку)

## Этапы создания проекта

1. Качаем и устанавливаем composer (перед этим php и т.д. и т.п.)

2. Переходим в папку, где будет проект и пишем

```
composer create-project laravel/laravel two (two - имя)
```

3. Устанавливаем dbal

```
composer require doctrine/dbal
```

4. Коннектимся к бд



5. Убираем защиту

A screenshot of a code editor showing the RouteServiceProvider.php file from a Laravel project named 'two'. The file contains PHP code for defining routes. The code includes annotations like '@var string' and 'RouteServiceProvider.php'.

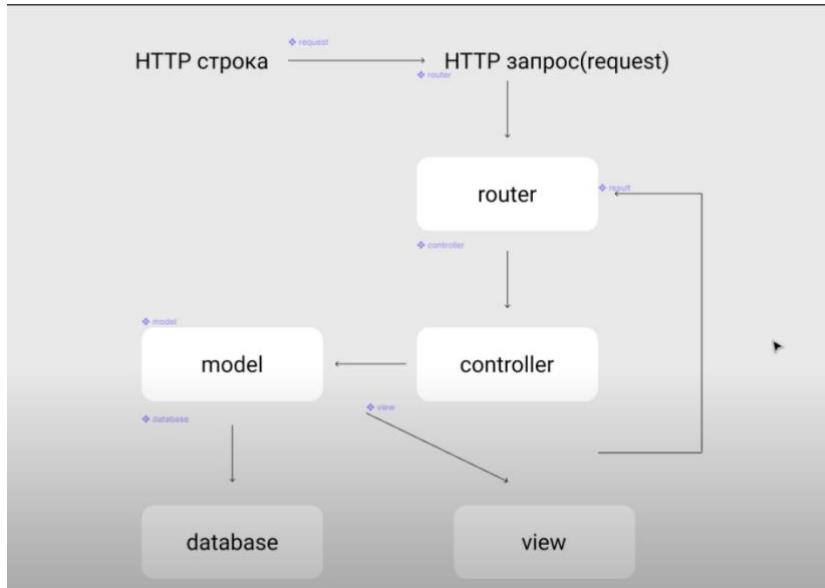
## Model-View-Controller

Model-View-Controller — схема разделения (шаблон проектирования) данных приложения и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо. Модель предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.

MVC паттерн – повторимая конструкция решения конкретной проблемы проектирования

Такой пример – работа абсолютно любого сайта основанного на фреймворках. Именно так принято разрабатывать сайты.

Такой паттерн – победа над сложностью за счет разделения труда



**Модель** – ПРЕДСТАВЛЕНИЕ одного объекта для интерфейса php. Когда мы берем какой-то пост из бд, то аргументы сохраняются в результирующем объекте. И из одной модели можно будет через синтаксис php обращаться к аргументам объекта (название цена и т.д. и т.п.). У НАС БУДЕТ МОДЕЛЬКА СО СВ-МИ КОТОРЫЕ МОЖНО БРАТЬ

Все это – классы. И в разном классе разная логика

Контроллер – Мастер обработки http запросов.

Модель – реализация действий с объектами бд.

Роутер – список запросов и соответствий.

Этапы:

1. **HTTP строка → HTTP запрос (request).** Тут наш сайт по протоколу обращается к нашему серверу и говорит ему что нужно что-то достать
2. **HTTP запрос (request) → router(маршрутизатор).** Маршрутизатор работает как администратор в отеле, который при запросе ключа от комнаты ищет его на полках и дает ключ. Тут же наш роутер обращается к контроллеру, у которого просит принести ему этот ключ

3. **router(маршрутизатор) → контроллер.** Контроллер по приказу роутера идет к модели и просит уже его достать этот ключ в любом виде из базы данных.
4. **Контроллер – модель.** Модель обращается в базу данных за определенным ключом и выдает его контроллеру
5. **Контроллер - view.** На этом этапе контроллер подготавливает данных в пригодный вид и выдает результат роутеру
6. **Router – result.** Роутер выводит результат на сайт

Все это можно сделать и в 1 файле без такого разделения, но это ГЕМОРОЙ.  
То же самое что жить всей семьей в одной комнате, можно, но не нужно.  
**ТАКАЯ МОДЕЛЬ ПРАВИЛЬНАЯ И ЭФФЕКТИВНАЯ**

### Http строка

HTTP — протокол прикладного уровня передачи данных (отправка запросов на сервер и возврат страниц)

### Request(route)

```
C
MacBook-Pro-artem:first_proj muzica$ php artisan serve
Starting Laravel development server: http://127.0.0.1:8000
[Mon Jul 25 17:03:26 2022] PHP 8.1.8 Development Server (http://127.0.0.1:8000) started
[Mon Jul 25 17:03:32 2022] 127.0.0.1:50777 Accepted
[Mon Jul 25 17:03:32 2022] 127.0.0.1:50777 Closing
```

Данная команда запускает Laravel сервер и выдает окно прослушки действий

```
<?php

use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/
Route::get('/', function () {
    return view('welcome');
});
```

Routers-web.php – Список маршрутизаторов (все что мы вбиваем в командную строку и какой результат получим на запрос)

```
Route::get('/my_page', function(){
    return 'this is my page';
}); // класс route с приватным методом get(тип запроса) и аргументами uri адрес и
//анонимной функцией
```

В этот файл мы можем внести свой список маршрутизаторов при помощи данного класса

this is my page

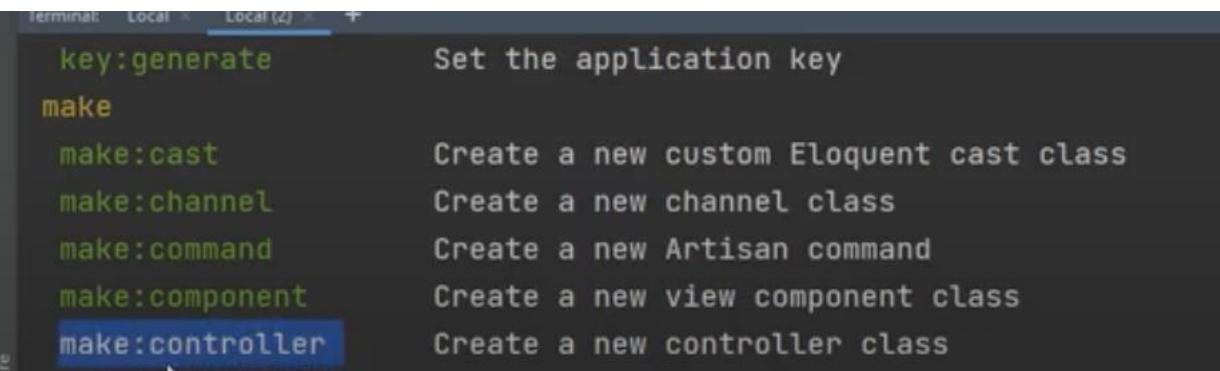
Роутер лишь маршрутизатор, который перенаправляет запросы.

## Controller

Все что находится внутри проекта Laravel – сайт, где все взаимосвязано. Каждая папка и каждый класс. ЭТО ПРАВИЛЬНАЯ СТРУКТУРА САЙТА.

Контроллер – Мастер обработки http запросов.

Контроллер – это класс.



```
Terminal: Local Local (2) +
key:generate      Set the application key
make
make:cast         Create a new custom Eloquent cast class
make:channel       Create a new channel class
make:command        Create a new Artisan command
make:component      Create a new view component class
make:controller     Create a new controller class
```

При помощи artisan можно создать контроллеры.

```
MacBook-Pro-artem:first_proj muzica$ php artisan make:controller -help
Description:
Create a new controller class

Usage:
make:controller [options] [--] <name>

Arguments:
name          The name of the class

Options:
--api          Exclude the create and edit methods from the controller.
--type=TYPE    Manually specify the controller stub file to use.
--force        Create the class even if the controller already exists
-i, --invokable Generate a single method, invokable controller class.
```

Документация.

```
php artisan make:controller MyPageController
```

ИМЕНА контроллеров должны быть с большой буквы и верблюжьей нотацией. Это конвенция (правило, договор)

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MyPageController extends Controller
{
    public function index(){
        return "My page";
    }

}
```

Вносим в наш новый контроллер(класс) новый метод, который мы сможем вызвать через роутер

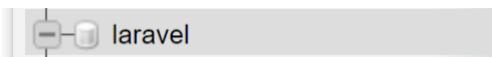
```
Route::get('/my_page', 'MyPageController@index');
```

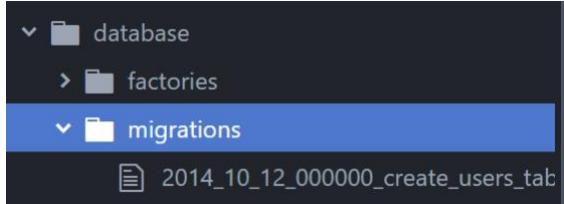
## Database (миграции)

Для работы с бд в laravel нужно создать миграцию для работы с контроллером

Миграция -> обычная таблица бд с колонками и типами данных

Атрибут - колонка. Все что хранится внутри таблицы - объекты, каждая колонка - атрибут объекта (колонки и атрибут синонимы)



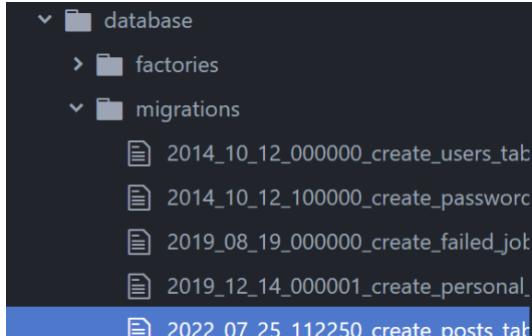


Команда для создания миграций [--] опции можно не писать

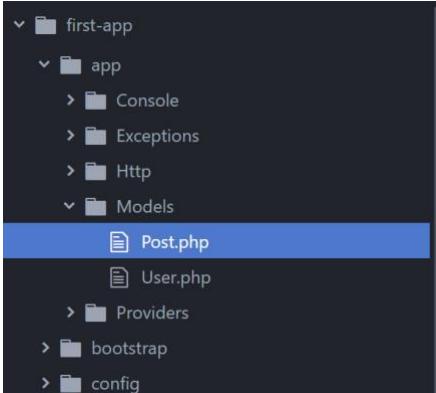
Для того чтобы не сойти с ума при создании миграции нужно сразу создавать модель, связанную с миграцией и ларавел создаст все по конвенции

```
PS C:\OpenServer\domains\laravel\first-app> php artisan make:model Post -m
```

Создание модели сразу с миграцией (имя с большой буквы и при связывании в конце Model не писать и обязательно в ед. числе)



Получаем миграцию (ключевое слово `create` в названии означает создание таблицы) и модель (ниже) и мы получаем даже класс с нашим именем



```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Factories\HasFactory;
6 use Illuminate\Database\Eloquent\Model;
7
8 class Post extends Model
9 {
10     use HasFactory;
11 }
```

```
/*
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('posts');
}
```

В нашей миграции в классе есть два метода `up` and `down` Нужны они для **накатывания и откатывания** миграций (так мы получаем и извлекаем данные)

При накатывании изменений все происходит в методе `up`  
А при откате в методе `down`

- Аналогия ктрл з ктрл у
- Аналогия бекапов (по сути, это они и есть)

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}
```

## РАЗБЕРЕМ МЕТОД АП

Здесь schema - класс фасад (удобная оболочка) далее обращаемся к методу класса create и аргументы функции (название таблицы и функция кол-бек) далее создаём айди (ключ в базе для отличия элементов) и две колонки (время занесения объекта в базу и время изменения элемента)

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->timestamps();
    });
}
```

В миграциях мы можем создавать свои колонки (нужно указать тип)

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('content');
        $table->unsignedBigInteger('likes')->nullable();
        $table->boolean('is_published')->default(1);
        $table->string('images')->nullable();
        $table->timestamps();
    });
}
```

ТУТ МЫ СОЗДАЛИ ТАБЛИЦУ

```
PDO::__construct()
PS C:\OpenServer\domains\laravel\first-app> php artisan migrate
```

Мигрируем таблицу (будет ошибка подключения к бд). Для этого в файле env (окружение (сторонние используемые программы)) подключаемся к бд

The screenshot shows the MySQL Workbench interface. On the left, there's a tree view of databases: information\_schema, laravel (which contains Новая, failed\_jobs, migrations, password\_resets, personal\_access\_tokens, posts, users), launcher, launcher2, mysql, and mutant. On the right, there's a table named 'migrations' with the following data:

		id	migration	batch
<input type="checkbox"/>		1	2014_10_12_000000_create_users_table	1
<input type="checkbox"/>		2	2014_10_12_100000_create_password_resets_table	1
<input type="checkbox"/>		3	2019_08_19_000000_create_failed_jobs_table	1
<input type="checkbox"/>		4	2019_12_14_000001_create_personal_access_tokens_ta...	1
<input type="checkbox"/>		5	2022_07_25_112250_create_posts_table	2

Получаем вот такую таблицу и бд. Тут есть таблица миграции, которая отслеживает все созданные таблицы для предотвращения повторной миграции

```
D:\lessons\first_project>php artisan migrate:rollback
```

Откат таблиц (поэтапно (нужно много раз откатывать все срашь он не откатит))

## Model

```
PS C:\OpenServer\domains\laravel\first-app> php artisan make:model Post -m
```

Создание модели сразу с миграцией (имя с большой буквы и при связывании в конце Model не писать и обязательно в ед числе)

С данными можно делать действия CRUD

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;
}
```

Структура модели

Наш класс Пост наследник класса модель (у него есть все св-ва и метода класса модель)

В классе модель описаны все с-ва и методы которые помогают работать с данными в таблице. Теперь мы можем работать с данными посредством модели (методами и св-ми класса пост который наследуется от класса модель)

Класс пост является МОДЕЛЬЮ, связанной с таблицей пост  
ХЕЛПЕРЫ

**Хелперы** - зарезервированные методы и функции в ларавел которые упрощают нам жизнь

```
class PostsController extends Controller
{
    public function posts(){
        $var = 'string';
        var_dump($var);
    }
}
```

(показывает какой тип данных и кол-ва символов)

string(6) "string" -- Функция php по дампингу (не хелпер)

```
public function posts(){
    $var = 'string';
    // var_dump($var);
    dd($var);
```

-- Хелпер (вывод данных и остановка выполнения)

```
"string" dd - dump die
```

## Модель чтения данных из базы

Существует множество способов чтения данных из таблицы, но мы пока что рассмотрим только 1

### Метод FIND

Для чтения через этот метод мы сначала обращаемся к классу и его статическому методу find(id)

```
$post = Post::find(3); // айди элемента в таблице  
Dd($post)
```

Переменная называется так же, как и класс

```
App\Models\Post {#620 ▾  
  #connection: "mysql"  
  #table: "posts"  
  #primaryKey: "id"  
  #keyType: "int"  
  +incrementing: true  
  #with: []  
  #withCount: []  
  +preventsLazyLoading: false  
  #perPage: 15  
  +exists: true  
  +wasRecentlyCreated: false  
  #escapeWhenCastingToString: false  
  #attributes: array:8 [▶]  
  #original: array:8 [▶]  
  #changes: []  
  #casts: []  
  #classCastCache: []  
  #attributeCastCache: []  
  #dates: []  
  #dateFormat: null  
  #appends: []  
  #dispatchesEvents: []  
  #observables: []  
  #relations: []  
  #touches: []  
  +timestamps: true  
  #hidden: []  
  #visible: []  
  #fillable: []  
  #guarded: array:1 [▶]  
}
```

Так дампится ОБЪЕКТ

Метод find создает ОБЪЕКТ МОДЕЛИ на основании класса

```
class PostController extends Controller
{
    public function index()
    {
        $post = new Post();
        dd($post);
    }
}
```

По сути, это аналог вот такого создания метода, но круче

```
#attributes: array:8 [▼
  "id" => 3
  "title" => "firstpost"
  "content" => "content"
  "likes" => 10
  "is_published" => 1
  "images" => "img"
  "created_at" => null
  "updated_at" => null
]
```

Здесь хранятся наши атрибуты (колонки) таблицы (сохранились в объект в ассоциативном массиве)

```
$post = Post::find(3); // айди элемента в таблице
dd($post->title);
```

Также можно обратиться к СВОЙСТВУ через объект (НАЗВАНИЕ КОЛОНКИ)

```
"firstpost"
```

РАЗБЕРЕМ СВ-ВА нашего объекта

```
App\Models\Post {#620 ▾
  #connection: "mysql"
  #table: "posts"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:8 [▶]
  #original: array:8 [▶]
  #changes: []
  #casts: []
  #classCastCache: []
  #attributeCastCache: []
  #dates: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  #hidden: []
  #visible: []
  #fillable: []
  #guarded: array:1 [▶]
}
```

Connection - тип соединения

Table - название таблицы

PrimaryKey - первичный ключ(айди)

Keytype - тип ключа

Incrementing - авто инкрементирование

```
class Post extends Model
{
    use HasFactory;
    protected $table = 'posts';
}
```

Явное указание типа таблицы (ЛУЧШЕ ДОПОЛНИТЕЛЬНО УКАЗАТЬ НАЗВАНИЕ ХОТЬ ОНО УЖЕ И СТОИТ)

Дз сделать миграции таблицы и тд и тп и вытянуть посредством find

---

ЭТАПЫ РАБОТЫ С БД (создание и чтение)

Создаем контроллер(для обращения к моделям) -> создаем функцию контроллера(пока пустую) -> создаем модель с миграцией(чтобы работать с базой данных) -> В миграцию создаём колонки таблицы ->

мигрируем(переносим в бд) -> В контроллере в функции создаем переменную с именем таблицы и обращаемся к классу созданной модели через метод модели find -> обращаемся к роутеру для вывода

## Строение Model

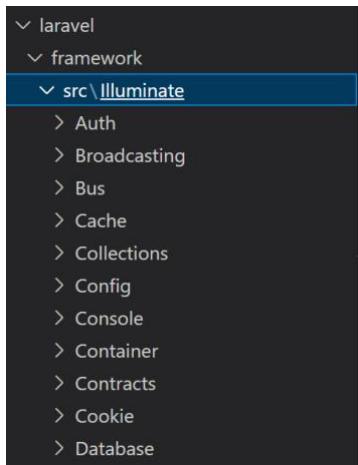
Когда мы создаем модель то мы создаем какой-то класс наследник класса модель

Сама же модель тоже наследует кучу классов

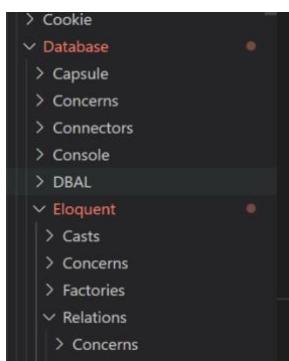
```
use ArrayAccess;
use Illuminate\Contracts\Broadcasting\HasBroadcastChannel;
use Illuminate\Contracts\Queue\QueueableCollection;
use Illuminate\Contracts\Queue\QueueableEntity;
use Illuminate\Contracts\Routing\UrlRoutable;
use Illuminate\Contracts\Support\Arrayable;
use Illuminate\Contracts\Support\CanBeEscapedWhenCastToString;
use Illuminate\Contracts\Support\Jsonable;
use Illuminate\Database\ConnectionResolverInterface as Resolver;
use Illuminate\Database\Eloquent\Collection as EloquentCollection;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
use Illuminate\Database\Eloquent\Relations\Concerns\AsPivot;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;
use Illuminate\Database\Eloquent\Relations\Pivot;
use Illuminate\Support\Arr;
use Illuminate\Support\Collection as BaseCollection;
use Illuminate\Support\Str;
use Illuminate\Support\Traits\ForwardsCalls;
use JsonSerializable;
use LogicException;
```

Синие - самые интересные они работают с бд

Все эти классы хранятся в папке **vendor** (там лежит содержимое всех зависимостей и отдельных модулей)



Отсюда мы уже и подтягиваем методы типа find



## ELOQUENT

Это тоже самое что и database, но он содержит красивые sql запросы (более простые и естественные, и удобные в использовании) (они приближены к php)

Также он содержит модели, которые уже имеют привязку к таблице (в отличии от database)

Вместо

```
SELECT * FROM first_project.posts WHERE id = 1;
```

Будет

```
$post = Post::find(1);
```

## DATABASE

Это уже более приближенное к sql (более низкоуровневый)

```
$user = DB::table('users')->find(3);
```

## Методы чтения данных (retrieve)

Все действия с бд можно прочитать в документации в разделе eloquent - getting start

```
class PostsController extends Controller
{
    public function posts () {
        $table = 'posts';
        $posts = Post::all();
        dd($posts);
    }
}

class Post extends Model
{
    use HasFactory;
    protected $table = 'posts';
}
```

(имя указывать как тут, а не как выше)

Вытаскиваем не 1 объект, а все

```
Illuminate\Database\Eloquent\Collection {#980 ▾
  #items: array:2 [▼
    0 => App\Mode...\\Post {#1227 ▶}
    1 => App\Mode...\\Post {#1228 ▶}
  ]
  #escapeWhenCastingToString: false
}
```

Collection - аналог массивов НО В КОНТЕКСТЕ ЛАРАВЕЛ (альтернатива массивам) Объекты тут устроены определенном образом с которыми можно работать множеством методов

```
contains
diff
except
find
fresh
intersect
load
loadMissing
modelKeys
makeVisible
makeHidden
only
toQuery
unique
```

(это не все)

Некоторые компании сразу говорят, что будет идти работа с коллекциями и методами коллекций

Если у массивов PHP есть методы, то аналог таких же методов есть и у коллекций

```
$table->string('name');  
$table->unsignedInteger('old')->nullable();
```

это коллекции (можно вызывать по цепочке через стрелочку)

ИСПОЛЬЗОВАНИЕ КОЛЛЕКЦИЙ И ХОРОШО И ПЛОХО каждая компания выставляет свои требования

—

### Выборка элементов

```
class PostsController extends Controller  
{  
    public function posts () {  
        $table = 'posts';  
        $posts = Post::all();  
        foreach($posts as $post){  
            dump($post -> title);  
        }  
        // dd($posts);  
    }  
}
```

Пройдемся по заголовкам наших атрибутов

"firstpost"

"вызов"

```
class PostsController extends Controller  
{  
    public function posts () {  
        $table = 'posts';  
        $posts = Post::where('status', 1)->get();  
        foreach($posts as $post){  
            dump($post -> title);  
        }  
        // dd($posts);  
    }  
}
```

Вернет все атрибуты

МЕТОД ГЕТ или ПОСТ ВОЗВРАЩАЕТ КОЛЛЕКЦИЮ

SELECT `title` FROM `posts` WHERE 1 - SQL

```
public function posts () {
    $table = 'posts';
    $post = Post::where('status', 1)->first();
    dump($post->title);
    // dd($posts);
}
```

Выбор только первой встречи

```
class FilmsController extends Controller
{
    public function first_films(){
        $table = 'films';
        $films = Film::where('status', 1)->first();
        foreach($films as $film){
            dump($film -> title);
        }
    }

    public function all_films(){
        $table = 'films';
        $films = Film::where('status', 1)->get();
        foreach($films as $film){
            dump($film->title);
        }
        // $table = 'films';
        // $films = Film::all();
        // foreach($films as $film){
        //     dump($film->title);
        // }
    }
}
```

```

class FriendsController extends Controller
{
    public function all_friends () {
        $table = 'friends';
        $friends = Friend::all();
        foreach($friends as $friend){
            dump($friend -> name);
        } //Вывод всех друзей
    }

    public function yung_friends () {
        $table = 'friends';
        $friends = Friend::where('old', '<', 30)->get();
        foreach($friends as $friend){
            dump($friend -> name);
        } // вывод друзей которым меньше 30
    }

    public function first_friend(){
        $table = 'friends';
        $friend = Friend::where('old', 18)->first();
        dump($friend->name); // вывод первого друга которому 18
    }
}

```

## Создание объекта в бд (добавление)

Создание происходит посредством обращения к модели и ее методу

```

class Post extends Model
{
    use HasFactory;
    protected $table = 'posts';
    protected $fillable = ['title','discription','likes','status']; // разрешение внесения добавлений
    // protected $fillable = ['title']; // разрешает изменения только названия(того что укажем)
}

protected $guarded = []; // разрешение внесения добавлений (разрешает полное редактирование)

```

Для начала в модели нужно дать разрешение на внесение изменений в таблицу

```
Route::get('/posts/create', 'PostsController@create');
```

Далее создаем роутер (страницу для создания объектов таблицы)

```

class PostsController extends Controller
{
    public function create(){
        $postsArr = [
            [
                'title' => 'Пост про работу',
                'description' => 'работа супер',
                'likes' => '20',
                'status' => '1'
            ],
            [
                'title' => 'Пост про письки',
                'description' => 'люблю письки', // создаем объект таблицы
                'likes' => '0',
                'status' => '0'
            ]
        ];
        foreach($postsArr as $item){
            dump($item);
            Post::create($item);
        } // занесение в таблицу колонок из массива (циклом по ключу)
    }
}

```

Создаем action (метод) create у контроллера

Далее можно создать двумерный ассоциативный массив, где ключи - атрибуты таблицы бд и значения - данные атрибута

И далее циклом foreach пройтись по нашему массиву и при помощи метода create добавить в таблицу наши объекты

```

Post::create([ // указываем по какому атрибуту вставлять и модель будет понимать в какую колонку какое значение класть
    // можно написать ручками а можно вставить готовый массив
    // сначала нужно убрать защиту добавления в бд
    // protected $guarded = []; // разрешение внесения добавлений (в модель Post)
    // (разрешает полное редактирование всех колонок-атрибутов)
    // protected $fillable = ['title']; // разрешает изменения только названия(того что укажем)
]);
dd('created');

```

Можно писать атрибуты сразу в метод create (точно так же, как и в массив) и так тоже будет добавлять

(если еще раз попытаться создать тоже самое, то все создастся)

## Методы обновления данных(update)

```
Route::get('/posts/update', 'PostsController@update');
```

Создадим роут для обновления

```
public function update(){
    $post = Post::find(2);

    $post->update([
        'title' => 'post 2',
    ]);
    dd($post); // dd - прерывает программу а dump продолжает
}
```

Далее пропишем метод класса, где переменная пост берет в себя все данные с таблицы, где айди = 2 и далее мы вызываем у переменной (уже объекта таблицы) пост метод update

```
#changes: array:2 [▼
    "title" => "post 2"
    "updated_at" => "2022-07-27 09:35:14"
]
```

Получаем наши изменения

(если мы еще раз попытаемся обновить, то обновление не встанет)

```
public function update(){
    $post = Post::find(1);
    dump($post);
    $post -> update([
        'title' => 'post 1',
    ]);
}
```

## Метод удаления данных(delete) и soft delete

```
Route::get('/posts/delete', 'PostsController@delete');

public function delete(){
    $post = Post::find(1);
    dump($post);
    $post -> delete();
    dd('deleted');
}
```

Вот и все удаление

Ошибка при повторном удалении

## Error

### Call to a member function delete() on null

<http://127.0.0.1:8000/posts/delete>

!!! ТАК УДАЛЯТЬ НЕ КРУТО И НЕ НАДО !!!

Такое удаление считается неправильным и неграмотным

Удалять нужно С ВОЗМОЖНОСТЬЮ ВОССТАНОВЛЕНИЯ и информацией кто что удалил

## SOFT DELETE

Мягкое удаление - удаление произошло, но по факту запись осталась (но она ни где и никогда не вылезет)

```
    */
    public function up()
    {
        Schema::create('pets', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->unsignedinteger('old');
            $table->text('content')->nullable();
            $table->boolean('status')->default(1)->nullable();
            $table->timestamps();

            $table->softDeletes();
        });
    }
```

Для применения этого способа нужно в миграции нашей таблица прописать атрибут

```
class Pet extends Model
{
    use HasFactory;
    use SoftDeletes;
    protected $table = 'pets';
    protected $guarded = [];
}
```

Далее в привязанной модели прописываем ТРЕЙД (use -> trade)

Трейд - готовая реализация метода (готовый импортируемый код)

```
PS C:\OpenServer\domains\laravel\first-app> php artisan migrate:fresh
```

Далее обновляем миграцию (этим способом ларавел удалит все таблицы и заново их загрузит) (ПОТЕРЯЕМ ВСЕ ДАННЫЕ)

Параметры										
	← T →	▼	id	name	old	content	status	created_at	updated_at	deleted_at
<input type="checkbox"/>			1	barsik	11	cat	0	2022-07-28 00:01:25	2022-07-28 00:01:25	NULL

После этого в нашей таблице появилась новая колонка deleted\_at

Если в этой колонке есть какая-то data, то ларавел воспринимает ее как удаленная

ТЕПЕРЬ МОЖНО УДАЛЯТЬ ЗАПИСИ ОБЫЧНЫМ СПОСОБОМ

ВОТ ТАК УДАЛЯТЬ КРУТО И ТАК УДАЛЯТЬ НУЖНО ВСЕГДА

```
public function restore(){
    $pet = Pet::withTrashed()->find(2);
    dump($pet);
    $pet -> restore();
    dd('restored');
}
```

Метод для восстановления данных

## Комбинированные методы создания и обновления данных

Это гарантия реализации на уровне CRUD

```
// firstOrCreate
```

Бывают ситуации, когда нужно взять что-то из базы и если такого элемента нет, то создать его (проверка на дубликаты) (добавляем в базу только уникальные элементы по определенным критериям)

```

public function firstOrCreate(){
    $pet = Pet::find(1);

    // $anotherPet = [
    //     'name' => 'rusya',
    //     'old' => 7,
    //     'status' => 0,
    //     'content' => 'dog'
    // ];

    $pet = Pet::firstOrCreate([
        'name' => 'rusya2'
    ], [
        'name' => 'rusya2',
        'old' => 7,
        'status' => 0,
        'content' => 'animal'
    ]);
    dump($pet -> content);
    dd('finished');
}

```

Метод firstOrCreate принимает два массива: первый указывает атрибут, который будет искааться в таблице, и если он его находит, то выполняется first, если не находит, то create (ЧТОБЫ ЭЛЕМЕНТ СОЗДАЛСЯ НЕ ДОЛЖНО СУЩЕСТВОВАТЬ ЗАДАННЫХ ЗНАЧЕНИЙ)

```
// updateOrCreate
```

Обновление записей с проверкой на дубликаты (проверка записей на дубликаты если данные изменены, но ОПРЕДЕЛЕННЫЕ атрибуты совпадают то его обязательно обновить) (одинаковые заголовки у постов)

```

public function updateOrCreate(){
    $pet = Pet::find(4);

    $pet = Pet::updateOrCreate([
        'name' => 'rusya2'
    ], [
        'name' => 'rusyaCREATE',
        'old' => 7,
        'status' => 0,
        'content' => 'not'
    ]);
}

```

Если он нашел объект с именем `rusya2`, то он его ОБНОВЛЯЕТ, а если не нашел, то СОЗДАЕТ

## Миграции. Редактирование миграций

Миграции по сути - таблицы.

Создание миграции - создание таблицы и тд

Про нейминг и создание читать в блоке модели

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateLangsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('langs', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('content')->nullable();
            $table->unsignedInteger('old')->nullable()->default[0];
            $table->unsignedInteger('status')->nullable();

            $table->timestamps();
            $table->softDeletes();
        });
    }
}
```

Метод `up` отвечает за внос изменений

```
public function down()
{
    Schema::dropIfExists('langs');
```

Down за бекап или откат

Существует 2 вида миграций:

Миграции создания

Миграции изменения (редактировать атрибут, добавить новый, изменить, удалить атрибут и дропнить таблицу (просто удалить через админку нельзя тк миграция существует, и она при деплое внедряется))

## Создание колонок

При создании миграции редактирования создавать при этом связанную модель не надо

Именовать такие таблицы принято add\_column\_...\_to\_posts\_table  
delte\_column\_... \_to\_friends\_table update\_column\_...

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AddColumnsToPostsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            //
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            //
        });
    }
}
```

При таком нейминге сразу laravel задает обращение к указанной таблице

НУЖНО ПРОПИСЫВАТЬ КАК НАКАТЫВАНИЕ, ТАК И ОТКАТЫВАНИЕ  
ОБЯЗАТЕЛЬНО  
ТАКЖЕ ДЕЛАТЬ ПРОВЕРКУ (НАКАТ-ОТКАТ-НАКАТ)

```
class AddColumnsToPostsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->text('content')->nullable(); // обязательно нулабл
        });
    }
}
```

Прописываем добавление случ образом С ОБЯЗАТЕЛЬНЫМ НУЛАБЛ (потому что если таблица уже существует и в старых версиях нет этой колонки, то будет жопа)

```
    ,
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->dropColumn(['content']);
        });
    }
}
```

Далее обязательно прописываем как мы будем откатывать

```
D:\Lessons\first_project>php artisan migrate      накатывание
D:\Lessons\first_project>php artisan migrate:rollback    откатывание
```

```
public function up()
{
    Schema::table('posts', function (Blueprint $table) {
        $table->text('content')->nullable()->after('title'); // обязательно нулабл
    });
}
```

Так можно накатить 'после'

## Удаление колонок

```
> php artisan make:migration delete_column_discscription_to_posts_table
```

```
    /*
     * ...
     */
    public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->dropColumn('discscription');
        });
    }
```

Удаление

```
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->text('discscription')->nullable()->after('content');
        });
    }
```

Откат (было удаление стало создание (тоже с нулабл))

```
php artisan migrate  
discscription_to_posts
```

## Редактирование колонок

### ПЕРЕИМЕНОВАНИЕ КОЛОНКИ

```
discscription_to_posts_table (51.3ms)
php artisan make:migration edit_column_content_to_posts_table
```

```
    /*
     * ...
     */
    public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->renameColumn('content', 'postscontent');
        });
    }
```

```
    /*
     * ...
     */
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->renameColumn('postscontent', 'content');
        });
    }
```

```
PS C:\OpenServer\domains\laravel\first-app> php artisan migrate
Migrating: 2022_07_29_122310_edit_column_content_to_posts_table

Error

Class 'Doctrine\DBAL\Driver\AbstractMySQLDriver' not found

at C:\OpenServer\domains\laravel\first-app\vendor\laravel\framework\src\Illuminate\Database\PDO\MySQLDriver.php:8
    4
    5     use Doctrine\DBAL\Driver\AbstractMySQLDriver;
    6     use Illuminate\Database\PDO\Concerns\ConnectsToDatabase;
    7
→  8 class MySQLDriver extends AbstractMySQLDriver
    9 {
    10     use ConnectsToDatabase;
    11 }
    12

1  C:\OpenServer\domains\laravel\first-app\vendor\composer\ClassLoader.php:571
   include()

2  C:\OpenServer\domains\laravel\first-app\vendor\composer\ClassLoader.php:428
   Composer\Autoload\includeFile()
PS C:\OpenServer\domains\laravel\first-app>
```

Активация Windows

Чтобы активировать Windows, перейдите

## УВИДИМ ТАКУЮ ОШИБКУ

Для решения этой проблемы нужно установить расширение DBAL в композере

```
composer require doctrine/dbal
```

## ИЗМЕНЕНИЕ ТИПА ДАННЫХ

```
php artisan make:migration change_column_postcontent_string_to_posts_table
```

```
public function up()
{
    Schema::table('posts', function (Blueprint $table) {
        $table->string('posts_content')->change();
    });
}
```

```
public function down()
{
    Schema::table('posts', function (Blueprint $table) {
        $table->text(['posts_content'])->change();
    });
}
```

## УДАЛЕНИЕ ТАБЛИЦЫ

```
public function up()
{
    Schema::dropIfExists('posts');
}
```

```
public function down()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->string('postscontent');
        $table->unsignedInteger('likes')->nullable();
        $table->boolean('status')->default(1);
        $table->timestamps();
    });
}
```

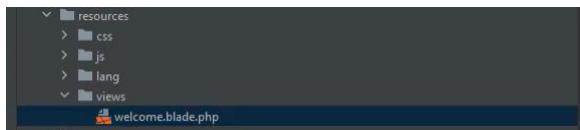
```
D:\lessons\first_project>php artisan migrate:Fresh
Dropped all tables successfully.
Migration table created successfully.
```

Перезагрузка всех таблиц

## View

### Знакомство

VIEW - шаблон куда мы заморачиваем данные для сайта



.blade.php - синтаксис blade под php

Blade упрощает написание php в html тк по сути это html теги с удобным синтаксисом php

```
@if (Route::has('login'))
|   <div class="hidden fixed top-0 right-0 px-6 py-4 sm:flex sm:justify-end sm:items-center sm:gap-2" style="background-color: #fff; border-bottom: 1px solid #e0e0e0; backdrop-filter: blur(10px);"
|     @auth
|       <a href="{{ url('/home') }}" class="text-sm text-gray-700 font-medium rounded-md py-1 px-3 border border-gray-300 hover:bg-gray-100 transition duration-150 ease-in-out">Home
|     @else
|       <a href="{{ route('login') }}" class="text-sm text-gray-700 font-medium rounded-md py-1 px-3 border border-gray-300 hover:bg-gray-100 transition duration-150 ease-in-out">Login
|       @if (Route::has('register'))
|         <a href="{{ route('register') }}" class="text-sm text-gray-700 font-medium rounded-md py-1 px-3 border border-gray-300 hover:bg-gray-100 transition duration-150 ease-in-out">Register
|       @endif
|     @endauth
|   </div>
@endif
```

Пример .blade.php

Наш роут обладает функцией callback (ответ на запрос)

```
public function index(){
    $posts = Post::all();
    return view('posts', compact('posts'));
}
```

Так мы возвращаем view (в аргументах только название файла без пути и без .blade.php)

```
<div>
  <?php foreach($posts as $post)>
</div>
```

Теперь вместо такого мы используем blade который все упрощает

```
@foreach($posts as $post)
|   <div>{{ $post -> title }}</div>
@endforeach
```

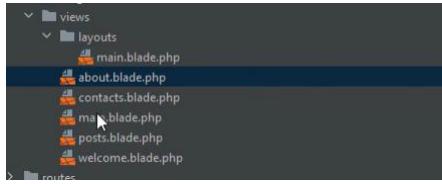
Теперь так

<!-- --> - комментарии

## Шаблоны view

При создании страниц сайта всегда что-то будет повторяться(например, структура) и чтобы не писать ее каждый раз во view есть ШАБЛОНИЗАЦИЯ

Мы переносим повторяющийся текст в отдельный файл в папку view-layouts и после можем к ней обращаться



```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Doc</title>
</head>
<body>
    @yield('doc')
</body>
</html>
```

В файл шаблона переносим повторяющийся код

```
@extends('layouts.main')
@section('doc')
    @foreach($posts as $post)
        <div>{{$post -> title}}</div>
    @endforeach
@endsection
```

В файл, где нужен этот код вставляем такую структуру

```
class PostsController extends Controller
{
    public function index(){
        $posts = Post::all();
        return view('posts', compact('posts'));
    }
}
```

Обязательно прописать для каждой страницы свой контроллер

```
Route::get('/posts/main', 'PostsController@main')->name('post.main');
Route::get('/posts/server1', 'PostsController@server1')->name('post.server1');
Route::get('/posts/server2', 'PostsController@server2')->name('post.server2');
Route::get('/posts/create', 'PostsController@create')->name('post.create');
```

Роутеру можно задать имя

```
@extends('layouts.main')
@section('doc')
    @foreach($server as $item)
        <h1>{{$item->title}}</h1>
        <p>{{$item->content}}</p>
    @endforeach
@endsection()
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div>
        <nav>
            <ul>
                <li><a href="{{route('post.main')}}">Main</a></li>
                <li><a href="{{route('post.server1')}}">Server 1</a></li>
                <li><a href="{{route('post.server2')}}">Server 2</a></li>
                <li><a href="{{route('post.create')}}">Create</a></li>
                <li><a href=""></a>Donate</li>
            </ul>
        </nav>
    </div>
    @yield('doc')
</body>
</html>
```

```

class PostsController extends Controller
{
    public function main(){
        return view('main');
    }

    public function create(){
        $postsArr = [
            'title' => 'magic',
            'content' => 'magic content',
            'likes' => 100,
            'status' => 1,
        ], [
            'title' => 'tech',
            'content' => 'tech content',
            'likes' => 10,
            'status' => 0,
        ];
        foreach($postsArr as $post){
            // dump($post);
            $mypost[] = Post::create($post);
        }
        return view('postsCreate', compact('mypost'));
    }
}

```

```

public function server1(){
    $server = Post::where('title','=','magic')->get();
    // dump($server);
    return view('server1', compact('server'));
}

public function server2(){
    $server = Post::where('title','=','tech')->get();
    return view('server2', compact('server'));
}

```

# Bootstrap в laravel

```
composer require laravel/ui
```

Для начала установим ui

```
laravel ui
ui
ui:auth          Scaffold basic login and registration views and routes
ui:controllers   Scaffold the authentication controllers
```

```
PS C:\OpenServer\domains\laravel\two\two> php artisan ui bootstrap
Bootstrap scaffolding installed successfully.
```

```
Please run "npm install && npm run dev" to compile your fresh scaffolding.
```

Далее устанавливаем bootstrap

```
npm install && npm run dev
```

Прописываем команду, которая подключит дополнительные css стили  
установит bootstrap  
(структура laravel css не всегда .css может быть и .scss)

Суть npm run dev - компиляция из формата scss и др. в css и собирает в папку public

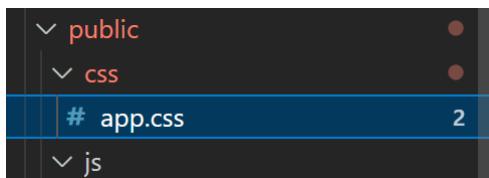
```
composer require laravel/ui
php artisan ui bootstrap
npm install
npm run dev
npx mix
```

ВЫПОЛНЯЕМ ВОТ ЭТИ КОМАНДЫ И ПОЛУЧАЕМ УСПЕХ

```
✓ Compiled Successfully in 7840ms
File      Size
/js/app.js 2.22 MiB
css/app.css 232 KiB
webpack compiled successfully
```

```
two > JS webpack.mix.js > ...
1 const mix = require('laravel-mix');
2
3 /*
4 | -----
5 | Mix Asset Management
6 | -----
7 |
8 | Mix provides a clean, fluent API for defining some Webpack build steps
9 | for your Laravel application. By default, we are compiling the Sass
10| file for the application as well as bundling up all the JS files.
11|
12 */
13
14 mix.js('resources/js/app.js', 'public/js')
15     .sass('resources/sass/app.scss', 'public/css')
16     .sourceMaps();
17
```

Тут можно все переделать



Все стили из bootstrap лежат в этой директории

Папка public нужна для обращения к ресурсам через публичный доступ(вопрос безопасности)

<link rel="stylesheet" href="{{ asset('css/app.css') }}> (грамотная  
сборка ссылки) {{PUBLIC\_PATH}} (asset - все доп. файлы проекта этот хелпер  
всегда попадает в нашу папку )  
Подключение bootstrap к файлу

## CRUD через интерфейс

Реализовывать будет связку frontend и backend

Всего для реализации crud в ларавел есть 7 шаблонов(actions)(рестфул контроллеры)

### # Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Хендлеры для crud через интерфейс(конвенция о названии роутеров и их запросов)

1. **INDEX** - перечень всех элементов (посты фотки и т.д. и т.п.) (главная страница)

```
Route::get('/servers', 'ServerController@index')->name('server.index'); //страница всех серверов
```

Роут для индекса

```
public function index(){
    $servers = Server::all();
    return view('server.index', compact('servers')); //вывод всех серверов
}
```

АКШОН для индекса

```
@extends('layouts.main')
@section('content')
<div>
    @foreach($servers as $server)
        <div>{$server->id}. {$server->title}</div>
    @endforeach
</div>
@endsection()
```

Вью для индекса

2. **CREATE** - отвечает за создание объектов бд

```
Route::get('/servers/create', 'ServerController@create')->name('server.create'); //страница создания сервера
```

```
public function create(){
    return view('server.create');
}
```

```
@extends('layouts.main')
@section('content')


<form action="{{route('server.store')}}" method="POST">
        @csrf
        <!-- СТАВИТ ЗАЩИТУ -->
        <div class="mb-3">
            <label for="title" class="form-label">Title</label>
            <input type="text" name="title" class="form-control" id="title" placeholder="Enter title">
        </div>
        <div class="mb-3">
            <label for="content" class="form-label">Content</label>
            <!-- <input type="password" class="form-control" id="exampleInputPassword1" -->
            <textarea class="form-control" name="content" id="content" cols="5" rows="5" placeholder="Enter content..."></textarea>
        </div>
        <div class="mb-3">
            <label for="image" class="form-label">Image</label>
            <input type="text" name="image" class="form-control" id="image" placeholder="Image URL...">
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
    </form>


@endsection()
```

3. **STORE** - обработчик форм методом post (для этого нужно добавить защиту @csrf)



419 | PAGE EXPIRED

Ошибка сайта без добавления защиты

## Межсайтовая подделка запроса (Cross-site request forgery)

CSRF — вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер, осуществляющий некую вредоносную операцию. Википедия

Отзыв

НУЖНО УЖЕ УМЕТЬ СОЗДАВАТЬ И ОБРАБАТЫВАТЬ ФОРМЫ(нейминг элементов такой же как название атрибутов в таблице бд)

```
Route::post('/servers', 'ServerController@store')->name('server.store'); //обработчик формы
```

Роут обработчик

```
public function store(){
    $data = request()->validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string'
    ]);
    // dump($data);
    Server::create($data);
    return redirect()->route('server.index');
}
```

Файл обработчик формы (ключи в массиве должны совпадать с именем бд в validate мы задаем ЗАЩИТУ говоря, что мы ожидаем и какой тип и только это) (если не указать ожидаемый тип может быть ошибка пустого поля)

```

<div>
    <form action="{{route('server.store')}}" method="POST">
        @csrf
        <!-- СТАВИТ ЗАЩИТУ -->
        <div class="mb-3">
            <label for="title" class="form-label">Title</label>
            <input type="text" name="title" class="form-control" id="title" placeholder="Enter title">
        </div>
        <div class="mb-3">
            <label for="content" class="form-label">Content</label>
            <!-- <input type="password" class="form-control" id="exampleInputPassword1"> -->
            <textarea class="form-control" name="content" id="content" cols="5" rows="5" placeholder="Enter content"></textarea>
        </div>
        <div class="mb-3">
            <label for="image" class="form-label">Image</label>
            <input type="text" name="image" class="form-control" id="image" placeholder="Image URL...">
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
    </form>
</div>
@endsection()

```

Форма для обработки(ОБЯЗАТЕЛЬНО С ЗАЩИТОЙ @csrf)(НЕЙМИНГ КАК В БД)

4. **SHOW** - Выцепить что-то конкретное (конкретный пост или конкретный сервер по айди или названию) (по методу гет)

`Route::post('/servers/{server}', 'ServerController@show')->name('server.show');` //обрабтчик формы  
Роут show (вместо {server} может быть все что угодно это просто условное обозначение)

```

public function show($id){
    $id = Server::findOrFail($id);
    dd($id->title);
}

```

(метод 1)

Экшон show FindOrFail - не выдает ошибку при ненаходе а перенаправляет на страницу 404

```

public function show(Server $id){
    dd($id->title);
}

```

(метод 2)

The screenshot shows a browser window with the URL `127.0.0.1:8000/servers/1`. The page content is a single line of text: `"magic craft"`.

Получаем в ответ

```
web.php           ServerController.php      show.blade.php X main.bl  
crud > resources > views > server > show.blade.php > div > div > h1  
1  @extends('layouts.main')  
2  @section('content')  
3  <div>  
4    <div><h1>{{$id->title}}</h1></div>  
5    <div>{{$id->content}}</div>  
6  </div>  
7  @endsection()
```

```
@extends('layouts.main')  
@section('content')  
<div>  
  @foreach($servers as $server)  
    <div><a href="{{route('server.show', [$server->id])}}>{{$server->id}}. {{$server->title}}</a></div>  
  @endforeach  
</div>  
@endsection()
```

Так мы осуществляем переход по ссылке с определенному серверу

## 5. EDIT - обработчик формы для обновления конкретного поста Update - patch обновление поста

```
Route::get('/servers/{id}/edit', 'ServerController@edit')->name('server.edit'); //обработчик формы  
Route::patch('/servers/{id}', 'ServerController@update')->name(['server.update']); //обработчик формы
```

```
public function edit(Server $id){  
    // dd($id->title);  
    return view('server.edit', compact('id'));  
}  
  
public function update(Server $id){  
    $data = request()->validate([  
        'title' => 'string',  
        'content' => 'string',  
        'image' => 'string'  
    ]);  
    $id -> update($data);  
    return redirect()->route(['server.show', $id->id]);  
}
```

```

@extends('layouts.main')
@section('content')


<form action="{{route('server.update', $id)}}" method="POST">
        @csrf
        @method('patch')
        <!-- СТАВИТ ЗАЩИТУ -->
        <div class="mb-3">
            <label for="title" class="form-label">Title</label>
            <input type="text" name="title" class="form-control" id="title" placeholder="Enter title" value="{$post->title}">
        </div>
        <div class="mb-3">
            <label for="content" class="form-label">Content</label>
            <!-- <input type="password" class="form-control" id="exampleInputPassword1"> -->
            <textarea class="form-control" name="content" id="content" cols="5" rows="5" placeholder="Enter content" value="{$post->content}">
        </div>
        <div class="mb-3">
            <label for="image" class="form-label">Image</label>
            <input type="text" name="image" class="form-control" id="image" placeholder="Image..." value="{$post->image}">
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
    </form>


@endsection()

```

## 6. DELETE - удаление поста

```

public function destroy(Post $post){
    $post->delete($post);
    return redirect()->route('post.index');
}

```

```

<form action="{{route('post.destroy', $post)}}" method="POST">
    @csrf
    @method('delete')
    <button type="button" class="btn btn-primary" href="{{route('post.destroy', $post->id)}}">Удалить</button>
</form>
<a href="{{route('post.index')}}">Назад</a>

```

## Отношения ОДИН КО МНОГИМ

Выражают отношения моделей(таблиц) в базе

УЖЕ НУЖНО ЗНАТЬ ПРО ОТНОШЕНИЯ

Один к одному применяется ооооооченб редко

Один ко многим - самое частое потом многое ко многому

-

Возьмем отношения ПОСТЫ - КАТЕГОРИИ (категория котики - много постов)

```

public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->timestamps();
    });
}

```

Создадим таблицу с категориями

```

Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('content');
    $table->unsignedInteger('likes')->nullable();
    $table->string('image')->nullable();
    $table->boolean('status')->default(1);
    $table->softDeletes();
    $table->timestamps();

    $table->unsignedBigInteger('category_id')->nullable(); // добавляем колонку категории

    $table->index('category_id', 'post_category_idx'); // добавляем колонку как индекс

    $table->foreign('category_id')->references('id')->on('categories'); // связываем таблицы
});

```

Привяжем таблицу с категориями к основной таблице(последние 3 строки)

The screenshot shows a database schema editor interface. A dropdown menu is open over a column named 'category\_id' in the 'posts' table. The menu contains two entries: 'cats - 1' and 'dogs - 2'. Below the menu, there is a checkbox labeled 'Игнорировать' (Ignore) which is checked. At the bottom of the interface, there are tabs for 'Столбец' (Column), 'Тип' (Type), 'Функция' (Function), 'Null' (Null), and 'Значение' (Value). The 'Значение' tab is currently selected.

Получаем связь атрибута таблицы posts к id таблице categories

```

 Illuminate\Database\Eloquent\Collection {#1026 ▶
 #items: array:2 [▼
   0 => App\Models\Category {#1028 ▶
     #connection: "mysql"
     #table: "categories"
     #primaryKey: "id"
     #keyType: "int"
     +incrementing: true
     #with: []
     #withCount: []
     #perPage: 15
     +exists: true
     +wasRecentlyCreated: false
     #attributes: array:4 [▼
       "id" => 1
       "title" => "cats"
       "created_at" => null
       "updated_at" => null
     ]
     #original: array:4 [▶]
     #changes: []
     #casts: []
     #classCastCache: []
     #dates: []
     #dateFormat: null
     #appends: []
     #dispatchesEvents: []
     #observables: []
     #relations: []
     #touches: []
     +timestamps: true
     #hidden: []
     #visible: []
     #fillable: []
     #guarded: array:1 [▶]
   ]
   1 => App\Models\Category {#1027 ▶}
 ]
}

public function index()
{
    $category = Category::find(1);

    $posts = Post::where('category_id', $category->id)->get();
    dd($posts);

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

```

ЭТО МОДЕЛЬ ОКАЗЫВАЕТСЯ

Выцепить все посты по категории

-- СПОСОБ 1

```

class Category extends Model
{
    use HasFactory;
    protected $guarded =[];
    protected $table = 'categories';

    public function posts(){
        return $this -> hasMany(Post::class, 'category_id', 'id');
        // $this - обращение к созданному на основе классу объекта
    }
}

```

Отношения

```

class Post extends Model
{
    use HasFactory;
    use SoftDeletes;

    protected $table = 'posts';
    protected $guarded = false;

    public function category()
    {
        return $this->belongsTo(related:Category::class, foreignKey: 'category_id', ownerKey: 'id');
    }
}

```

Обратное отношение

```

class PostController extends Controller
{
    public function index(){
        // $posts = Post::all();
        // return view('post.index', compact('posts'));

        $categ = Category::find(1);
        // // dump($categ->title);
        // $posts = Post::where('category_id', $categ->id)->get();
        // foreach($posts as $post){
        //     dd($post->title);
        // }

        dd[$categ->posts];
    }
}

```

Проверка

## Модификация crud - категории(один ко многим)

Создание:

```

public function create(){
    $category = Category::all();
    return view('post.create', compact('category'));
}

public function store(){
    $data = request()->validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'category_id' => ''
    ]);
    Post::create($data);
    return redirect()->route('post.index');
}

public function show(Post $post){
    return view('post.show', compact('post'));
}

```

Меняем create store

```

<div class="input-group mb-3">
    <label class="input-group-text" for="category">Категория</label>
    <select class="form-select" id="category" name="category_id">
        @foreach($category as $id)
            <option value="{{ $id->id }}>{{ $id->title }}</option>
        @endforeach
    </select>
</div>

```

Добавляем селект

Категория

cats

Создать

cats  
dogs  
spiders  
humans

## Результат

```
<div><p><b>Категория:</b>
{$post->category_id}</p></div>
```

Добавляем в show

SaintAdmin    Создать пост

# cats

catss

Картинка: catsss

Категория: 1

Статус: Опубликован

[Редактировать](#)  
[Назад](#)

[Удалить](#)

## Результат

Редактирование:

```
public function edit(Post $post){
    $category = Category::all();
    return view('post.edit', compact(['post', 'category']));
}
```

Меняем edit

```

public function update(Post $post){
    $data = request() -> validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'status' => 'boolean',
        'category_id' => 'integer'
    ]);

    $post->update($data);
    return redirect() -> route('post.index', $post->id);
}

```

Меняем update

```

<select class="form-select" id="category" required name="category_id">
    @foreach($category as $id)
        <option
            {{ $id->id === $post->category_id ? 'selected' : ''}}
            value='{{ $id->id }}'>{{ $id->title }}</option>
    @endforeach
</select>

```

Добавляем селект с проверкой (update)



Результат

## Отношения МНОГИЕ КО МНОГИМ

Реализуем как теги - посты

РАБОТАЕТ ЭТО НЕ КАК ОДИН КО МНОГИМ А ЧЕРЕЗ ТАБЛИЦУ ЗЕРКАЛО

```

php artisan make:model Tag -m
Model created successfully.
Created Migration: 2022_08_05_073110_create_tags_table

```

Таблица тегов

```

PS C:\OpenServer\domains\laravel\crudapi> php artisan make:model PostTag -m
Model created successfully.
Created Migration: 2022_08_05_073250_create_post_tags_table

```

Промежуточная таблица (имя из двух связь таблиц по алфавиту)

Можно не создавать модель, но лучше создать

- 🐘 2022\_08\_05\_024131\_create\_posts\_table.php
- 🐘 2022\_08\_05\_073110\_create\_tags\_table.php
- 🐘 2022\_08\_05\_073250\_create\_post\_tags\_table.php

## Таблицы

```
class CreateTagsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->timestamps();
        });
    }
}
```

Содержание таблицы тег можно удалить, а можно оставить(лучше оставить)

```
public function up()
{
    Schema::create('post_tags', function (Blueprint $table) {
        $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();
    });
}
```

Содержание таблицы ПостТег

```
> app > models > <-- Post.php > Post > tags  
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Factories\HasFactory;  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\SoftDeletes;  
  
class Post extends Model  
{  
    use HasFactory;  
    use SoftDeletes;  
    protected $table = 'posts';  
    protected $guarded = [];  
  
    public function tags(){  
        return $this->belongsToMany(Tag::class, 'post_tags', 'post_id', 'tag_id');  
    }  
}
```

Добавляем связь для Post

```
class Tag extends Model  
{  
    use HasFactory;  
  
    public function posts(){  
        return $this->belongsToMany(Post::class, 'post_tags', 'tag_id', 'post_id');  
    }  
}
```

Добавляем связь для Tag

```
class PostController extends Controller  
{  
    public function index(){  
        // $posts = Post::all();  
        // return view('post.index', compact('posts'));  
  
        $post = Post::find(1);  
        dd($post->tags);  
    }  
}
```

Проверка

По аналогии делается обратное

## Модификация crud - Теги(многие ко многим)

СПОСОБ 1 (ПЛОХОЙ) - НАРУШЕНИЕ СОХРАННОСТИ БАЗЫ (НЕТ ТРАНЗАКЦИЙ)

```
class PostTag extends Model
{
    use HasFactory;
    protected $guarded = false;
}
```

Снимаем защиту с зеркала

```
<div class="input-group mb-3">
<select class="form-select" multiple aria-label="multiple select example" name="tags[]>
    <option selected>Выберите теги(ctrl)</option>
    @foreach($tags as $tag)
        <option value="{{ $tag->id}}">{{ $tag->title}}</option>
    @endforeach
</select>
```

Добавляем опцию выбора мультиселект

```
public function store()
{
    $data = request()->validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'category_id' => 'integer',
        'tags' => 'array'
    ]);
    $tags = $data['tags'];
    unset($data['tags']);
    // dd($tags,$data);
    $post = Post::create($data);
    foreach ($tags as $tag) {
        PostTag::firstOrCreate([], // исключаем дублирование
            [
                'tag_id' => $tag,
                'post_id' => $post->id
            ]
        );
    }
    return redirect()->route('post.index');
}
```

Обрабатываем

СПОСОБ 2 - БОЛЕЕ ПРОФЕССИОНАЛЬНЫЙ, НО ВСЕ РАВНО БЕЗ ТРАНЗАКЦИЙ

Создание

```

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Post extends Model
{
    use HasFactory;
    use SoftDeletes;
    protected $table = 'posts';
    protected $guarded = [];

    public function tags(){
        return $this->belongsToMany(Tag::class, 'post_tags', 'post_id', 'tag_id');
    }
}

```

## Связь для постов

```

class Tag extends Model
{
    use HasFactory;

    public function posts(){
        return $this->belongsToMany(Post::class, 'post_tags', 'tag_id', 'post_id');
    }
}

```

## Связь для тегов

```

public function create(){
    $category = Category::all();
    $tags = Tag::all();
    return view('post.create', compact('category', 'tags'));
}

```

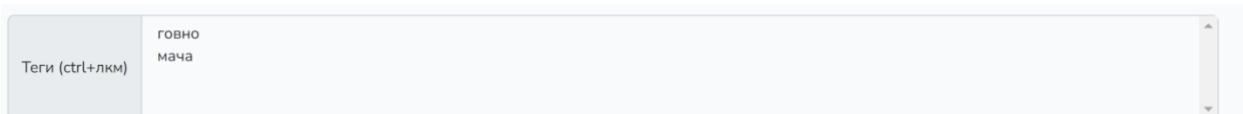
## Добавляем теги

```

<div class="input-group mb-3">
    <label class="input-group-text" for="tags">Теги (ctrl+лкм)</label>
    <select name="tags[]" class="form-select" id="tags" multiple aria-label="multiple select">
        @foreach($tags as $tag)
            <option value="{{ $tag->id }}">{{ $tag->title }}</option>
        @endforeach
    </select>
</div>

```

## Показываем окно тегов



```

public function store(){
    $data = request() -> validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'category_id' => 'integer',
        'tags' => ''
    ]);

    $tag = $data['tags'];
    unset($data['tags']);

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

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

    return redirect() -> route('post.index');
}

```

Редакчим store

Отображения пока не сделал

Редактирование

```

public function edit(Post $post){
    $category = Category::all();
    $tags = Tag::all();
    return view('post.edit', compact(['post', 'category', 'tags']));
}

```

Добавим в edit

```

public function update(Post $post)
{
    $data = request()->validate([
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'category_id' => 'integer',
        'tags' => ''
    ]);
    $tags = $data['tags'];
    unset($data['tags']);
    // dd($tags, $data);
    $post -> update($data);

    $post->tags()->sync($tags); // tags() - запрос в базу
    return redirect()->route('post.show', $post->id);
}

```

Меняем атак на синк чтобы не только добавляли новое, но и убирали старое(удаляет все до момента привязки и добавляет новое)

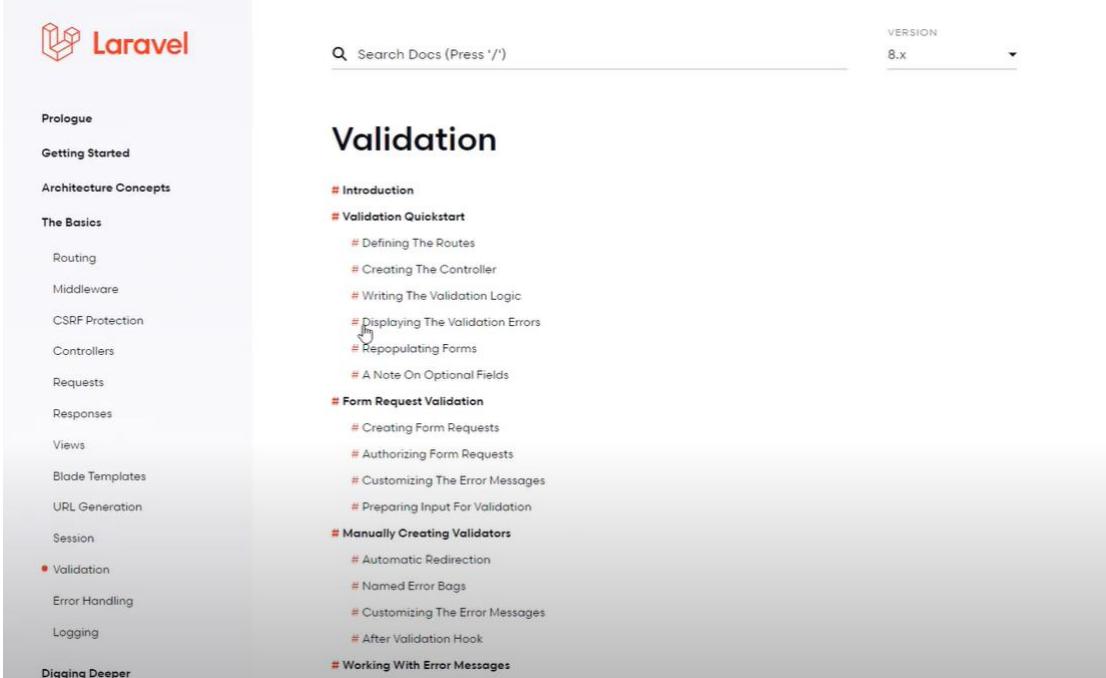
```

    </div>
    <div class="input-group mb-3">
        <select class="form-select" multiple aria-label="multiple select example" name="tags[]>
            <option disabled>Выберите теги(ctrl)</option>
            @foreach($tags as $tag)
                <option @foreach($post->tags as $postTag)
                    {$tag->id === $postTag->id ? 'selected' : ''}>
                    @endforeach
                    value="{{$tag->id}}>{{$tag->title}}
                </option>
            @endforeach
        </select>
    </div>
    <button type="submit" class="btn btn-primary">Изменить</button>

```

Добавляем селект

## Модификация crud - обработчик ошибок



The screenshot shows the Laravel documentation website at [laravel.com/docs/8.x/validation](#). The page title is "Validation". The sidebar on the left lists various documentation sections, and the main content area displays the "Validation" chapter with its sub-sections and links.

**Validation**

- # Introduction
- # Validation Quickstart
  - # Defining The Routes
  - # Creating The Controller
  - # Writing The Validation Logic
  - # Displaying The Validation Errors
  - # Repopulating Forms
  - # A Note On Optional Fields
- # Form Request Validation
  - # Creating Form Requests
  - # Authorizing Form Requests
  - # Customizing The Error Messages
  - # Preparing Input For Validation
- # Manually Creating Validators
  - # Automatic Redirection
  - # Named Error Bags
  - # Customizing The Error Messages
  - # After Validation Hook
- # Working With Error Messages
  - # Specifying Custom Messages In Configuration File

ПОЛЬЗУЕМСЯ ДОКУМЕНТАЦИЕЙ

Способ 1 - просто добавить атрибут required в html (не оч.)

Способ 2 - обработчик ошибок(круто если не стирать введенный текст при ошибке)

```
<div class="mb-3">
    <label for="title" class="form-label">Title</label>
    <input name="title" type="text" class="form-control" id="title" placeholder="Название">
    @error('title')
        <p class="text-danger">Ошибка ввода</p>
    @enderror
</div>
```

Добавляем error

```
@error('content')
<p class="text-danger">{{$message}}</p>
@enderror
```

Зарезервированная переменная выводит ошибку

```
$data = request()->validate([
    'title' => 'string',
    'content' => 'required|string',
    'image' => 'string',
    'category_id' => 'integer',
    'tags' => 'array'
]);
```

Так можно добавить обязательность ввода

The content field is required.

Результат

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

Чтобы введённый без ошибок текст остался

A screenshot of a web application interface. It shows two input fields: 'Title' and 'Content'. The 'Title' field contains the text 'dsfg'. The 'Content' field has the placeholder 'Содержание:' and displays the error message 'The content field is required.'.

```
<select class="form-control" id="category" name="category_id">
    @foreach($categories as $category)
        <option
            {{ old('category_id') == $category->id ? ' selected' : '' }}>
            {{ $category->title }}</option>
    @endforeach
</select>
```

Чтобы категория осталась

## Отношения один ко многим и многие ко многим через конвенцию Laravel

Можно кодить так как было описано выше, но это не ЧЕРЕЗ КОНВЕНЦИЮ

Через конвенцию можно сделать проще и круче

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->string('content');
        $table->string('image')->nullable();
        $table->boolean('status')->default(1);
        $table->unsignedBigInteger('category_id')->nullable();
        $table->timestamps();
        $table->softDeletes();

        $table->unsignedBigInteger('category_id');
    });
}
```

Создаем обычный атрибут, который будем связывать(имя должно совпадать с именем модели) (category - Category) (ед. числ)

```
> posts_table
> php artisan make:model Category
```

Создадим модель с привязкой (один ко многим)

```

Category.php
Post.php
User.php
> Providers
> bootstrap
> config
< database
    > factories
    < migrations
        Category.php
        Post.php
        User.php
        2014_10_12_000000_create_users_table.php
        2014_10_12_100000_create_password_rese...
        2019_08_19_000000_create_failed_jobs_tab...
        2019_12_14_000001_create_personal_acces...
        2022_08_08_093452_create_posts_table.php
        2022_08_08_093908_create_categories_table...
```

Получаем

```
class CreateCategoriesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->timestamps();
        });
    }
}
```

Содержание категории

```
: categories_table
: php artisan make:model Tag -m
Создаем теги (многие ко многим)
```

```
    /**
     * Run the migrations.
     */
    public function up()
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->timestamps();
        });
    }
}
```

Содержание тегов

```
PS C:\OpenServer\domains\laravel\relations2> php artisan make:migration create_post_tag_table --create
Created Migration: 2022_08_08_094437_create_post_tag_table
```

Для связи многие ко многим нужно создать отдельную МИГРАЦИЮ БЕЗ  
МОДЕЛИ(имя в ед. числе!!)

```

class CreatePostTagTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('post_tag', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }
}

```

Получаем

```

public function up()
{
    Schema::create('post_tag', function (Blueprint $table) {
        $table->id();

        $table->unsignedBigInteger('post_id');
        $table->unsignedBigInteger('tag_id');

        $table->timestamps();
    });
}

```

Содержание таблицы post\_tag

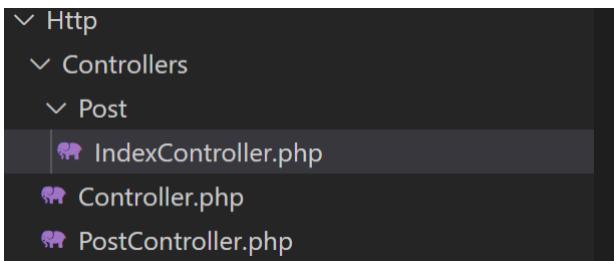
НА ЭТОМ ВСЕ ларавел уже все сам понимает(если правильно задать имена)

ЕСЛИ ВЫЛАЗИТ ОШИБКА, ТО НУЖНО УДАЛИТЬ ТАБЛИЦУ МИГРАЦИИ В БД И  
УДАЛИТЬ ТАБЛИЦУ

НЕ РАБОТАЕТ

## Однометодные контроллеры

Однометодные и многометодные контроллеры одинаково хорошо  
Просто считается, что однометодные контроллеры считаются более  
грамотными



Создаем папку, где будут наши однометодные контроллеры и именуем контроллеры по названию функции

```
<?php

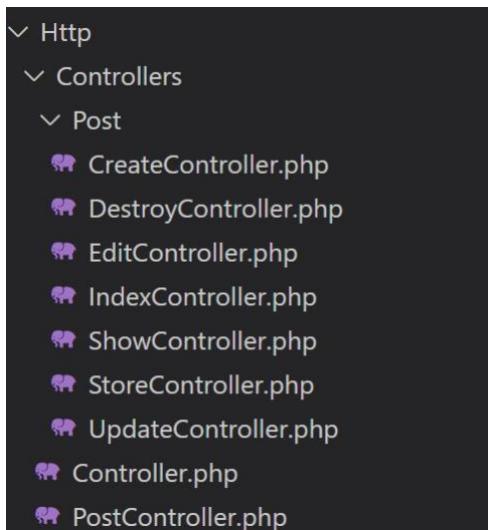
namespace App\Http\Controllers\Post;
use App\Http\Controllers\Controller;

use App\Models\Post;

class IndexController extends Controller
{
    public function __invoke(){
        $posts = Post::all();
        return view('post.index', compact('posts'));
    }
}
```

Очищаем контроллер и добавляем `__invoke` который в концепции ООП РНР выполняется первый и функцию можно никак не называть т. к. она - название файла

Также обязательно меняем `namespace` и `use`



Переносим все функции одного контроллера по разным контроллерам

```

<?php

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\Post;

Route::get('/', function () {
    return view('welcome');
});

Route::group(['namespace' => 'Post'], function(){
    Route::get('/posts', 'IndexController')->name('post.index');
    Route::get('/posts/create', 'CreateController')->name('post.create');
    Route::post('/posts', 'StoreController')->name('post.store');
    Route::get('/posts/{post}', 'ShowController')->name('post.show');
    Route::get('/posts/{post}/edit', 'EditController')->name('post.edit');
    Route::patch('/posts/{post}', 'UpdateController')->name('post.update');
    Route::delete('/posts/{post}', 'DestroyController')->name('post.destroy');
});|

```

## Содержание роутеров

## Класс Request

Основная задача этого класса проверка полученных данных на корректность и их передача дальше  
Используется для оптимизации

```

class StoreController extends Controller
{
    public function __invoke()
    {
        $data = request()->validate([
            'title' => 'required|string',
            'content' => 'string',
            'image' => 'string',
            'category_id' => '',
            'tags' => ''
        ]);
        $tags = $data['tags'];
        unset($data['tags']);

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

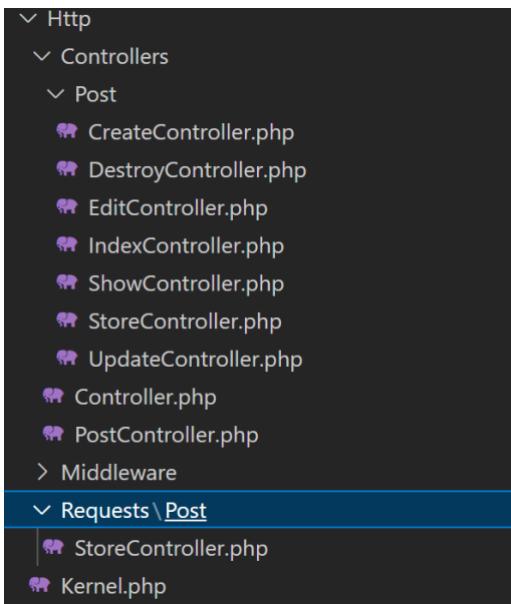
```

Такой код на профессиональной арене считается ПЛОХИМ И НЕПРОФЕССИОНАЛЬНЫМ

Тут мы используем метод `request()` но так делать тоже не оч. круто

```
php artisan make:request Post/StoreRequest
```

Создаем реквест вот так (имя реквеста - имя контроллера, где будет реквест)



```
public function authorize()
{
    return true;
}
```

В реквесте меняем FALSE НА TRUE тем самым включая его

```
public function rules()
{
    return [
        'title' => 'string',
        'content' => 'string',
        'image' => 'string',
        'category_id' => 'integer',
        'tags' => ''
    ];
}
```

Добавляем правила

```
<?php

namespace App\Http\Controllers\Post;
use App\Http\Controllers\Controller;
use App\Http\Requests\Post\StoreRequest;
use App\Models\Post;

class StoreController extends Controller
{
    public function __invoke(StoreRequest $request){
        $data = $request -> validated();

        $tag = $data['tags'];
        unset($data['tags']);

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

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

        return redirect() -> route('post.index');
    }
}
```

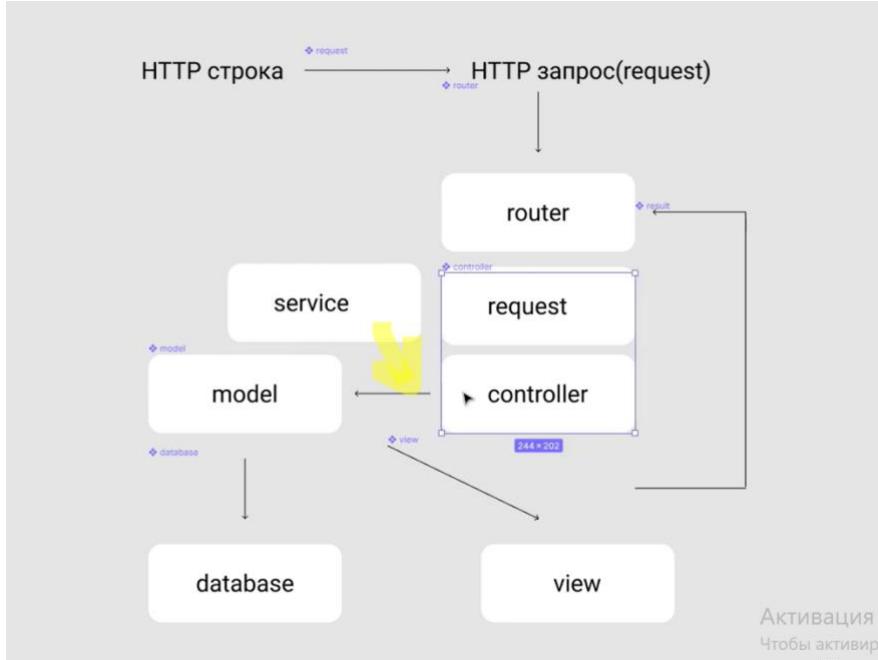
Меняем контроллер

ТАК МЫ КРУТО ОПТИМИЗИРОВАЛИ КОД

## Класс Service

Берет на себя обязанности по CRUD

Логика - берем алгоритм работы с бд который связан с отдачей результата и бд



## SERVICE - ПРОСЛОЙКА МЕЖДУ КОНТРОЛЛЕРОМ И МОДЕЛЬЮ

Прямая обязанность - грамотное взаимодействие с бд

Сервис помогает контроллеру взаимодействовать с моделью

```
class StoreController extends Controller
{
    public function __invoke(StoreRequest $request)
    {
        $data = $request->validated();
        $tags = $data['tags'];
        unset($data['tags']);

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

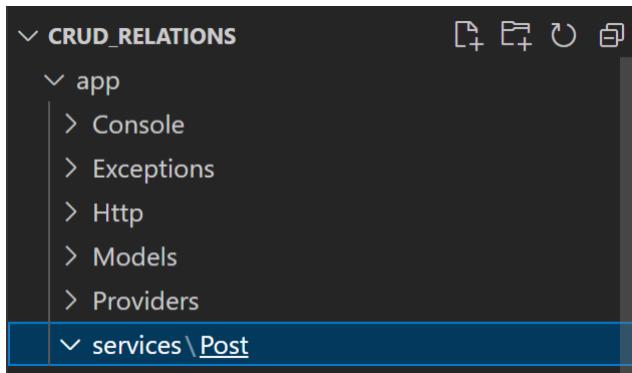
        $post->tags()->attach($tags);

        return redirect()->route('post.index');
    }
}
```

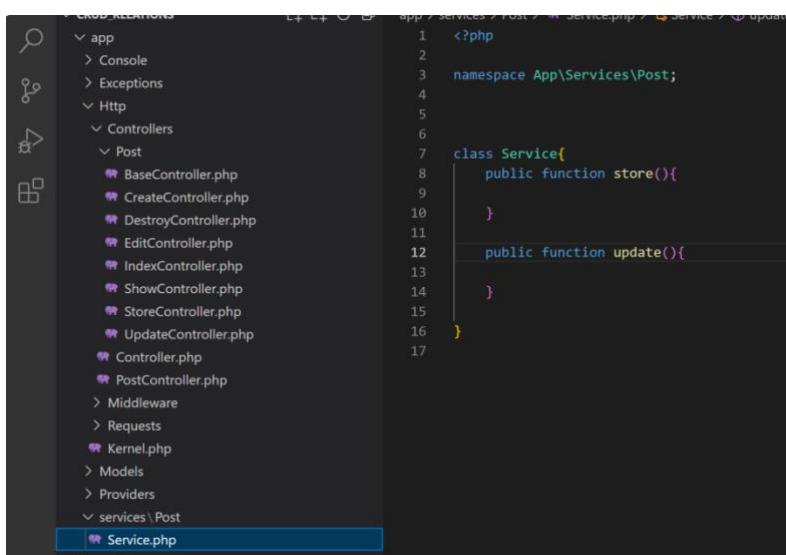
На самом деле вот такой код - нарушение норм (в будущем будет мусор и много путаницы)

По-хорошему нужно вынести работу с моделью в отдельное место (мы уже вынесли валидацию)

Сервис - концепт не ларавел а общепринятый концепт разработки на PHP



Создаем папку на территории app



Создаем в папке класс

```
namespace App\Http\Controllers\Post;  
  
use App\Services\Post\Service;  
  
class BaseController  
{  
    public $service;  
  
    public function __construct(Service $service)  
    {  
        $this->service = $service;  
    }  
}
```

Создаем базовый контроллер для подключения сервиса к контроллерам СО СВОЙСТВОМ И КОНСТРУКТОРОМ

1 присвоили свойству базового контроллера экземпляр на основании класса Service(стал объектом на основании класса)

2 создаем объект от класса сервис и присваиваем его в переменную

```
class BaseController extends Controller
{
    public $service;

    public function __construct(Service $service){
        $this->service = $service;
    }
}
```

Прописываем базовому контроллеру наследование от класса контроллера

```
class CreateController extends BaseController
{
    public function __invoke()
    {
        $category = Category::all();
        $tags = Tag::all();
        return view('post.create', compact('category', 'tags'));
    }
}
```

Меняем наследование в контроллеров

```
<?php

namespace App\Services\Post;
use App\Models\Post;

class Service{
    public function store($data){

        $tag = $data['tags'];
        unset($data['tags']);

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

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

    public function update(){

    }
}
```

Содержание класса сервис

```
class StoreController extends BaseController
{
    public function __invoke(StoreRequest $request){
        $data = $request -> validated();

        $tag = $data['tags'];
        unset($data['tags']);

        $this->service->store()

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

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

        return redirect() -> route('post.index');
    }
}
```

Теперь можно вызвать класс сервис и у него метод стор

```
class StoreController extends BaseController
{
    public function __invoke(StoreRequest $request){
        $data = $request -> validated();
        $this->service->store($data);
        return redirect() -> route('post.index');
    }
}
```

Меняем наш контроллер

КОНТРОЛЛЕР - МЕСТО ПЕРЕСЕЧЕНИЯ МНОГИХ ЛОГИК ПОЭТОМУ ЕГО НЕ  
НУЖНО НАГРУЖАТЬ ЕГО НУЖНО РАСПРЕДЕЛЯТЬ ПО РЕКВЕСТАМ СЕРВИСАМ И  
ТД

## Классы Factory и Seed

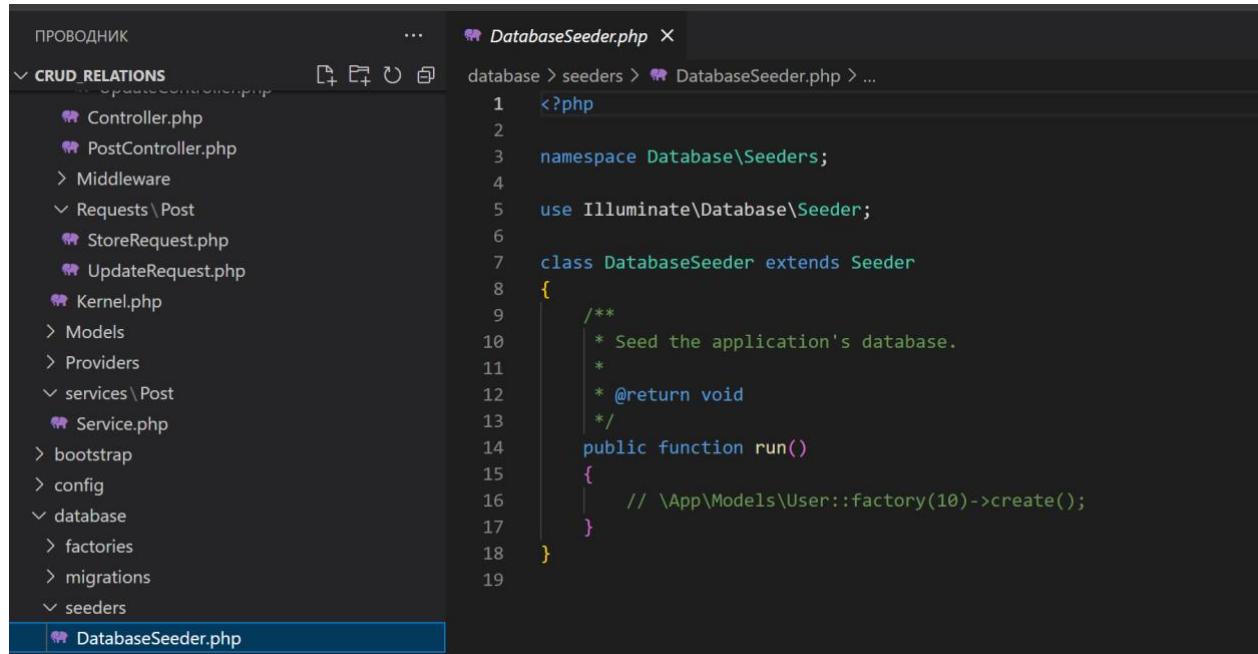
Деплой - Развёртывание программного обеспечения — это все действия, которые делают программную систему готовой к использованию. Данный процесс является частью жизненного цикла программного обеспечения.

Деплой — процесс «разворачивания» веб-сервиса, например, сайта, в рабочем окружении. Рабочее окружение — место, где сайт запускается и доступен для запросов. Это может быть как готовый хостинг, так и своя

собственная серверная инфраструктура. Деплоятся не только веб-сервисы, но любые сервисы, доступные по сети.

## Класс seed

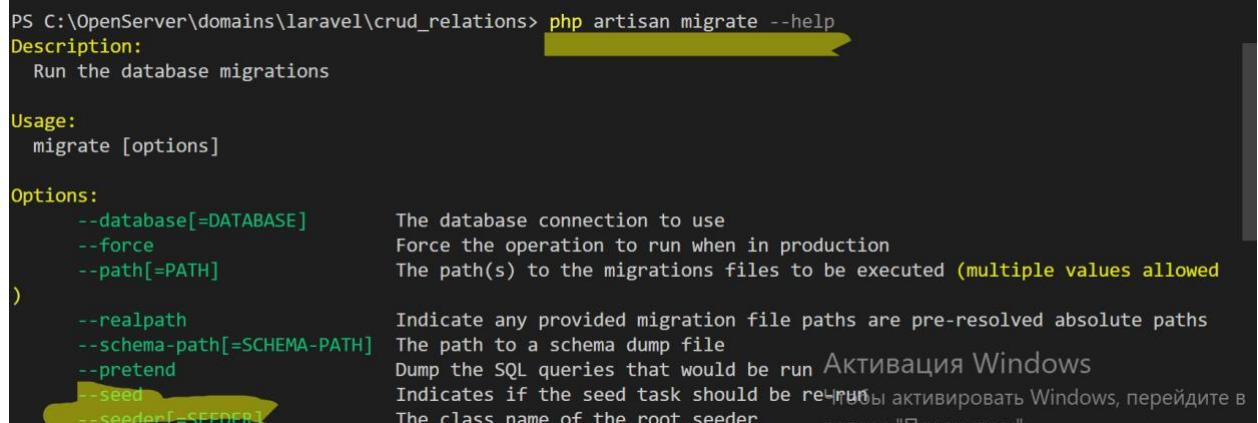
Нужен для развертывания каких-либо данных при деплое на сервер



```
ПРОВОДНИК ... DatabaseSeeder.php ×
CRUD_RELATIONS database > seeders > DatabaseSeeder.php > ...
Controller.php
PostController.php
Middleware
Requests\Post
StoreRequest.php
UpdateRequest.php
Kernel.php
Models
Providers
services\Post
Service.php
bootstrap
config
database
factories
migrations
seeders
DatabaseSeeder.php

1 <?php
2
3 namespace Database\Seeders;
4
5 use Illuminate\Database\Seeder;
6
7 class DatabaseSeeder extends Seeder
8 {
9     /**
10      * Seed the application's database.
11      *
12      * @return void
13      */
14     public function run()
15     {
16         // \App\Models\User::factory(10)->create();
17     }
18 }
```

Расположение и содержание



```
PS C:\OpenServer\domains\laravel\crud_relations> php artisan migrate --help
Description:
Run the database migrations

Usage:
migrate [options]

Options:
--database[=DATABASE]           The database connection to use
--force                           Force the operation to run when in production
--path[=PATH]                     The path(s) to the migrations files to be executed (multiple values allowed)
)
--realpath                        Indicate any provided migration file paths are pre-resolved absolute paths
--schema-path[=SCHEMA-PATH]       The path to a schema dump file
--pretend                          Dump the SQL queries that would be run
--seed                            Indicates if the seed task should be run
--seeder[=SEEDER]                 The class name of the root seeder
```

Как найти команду (позволяет запустить все что находится в методе)

```
public function run()
{
    echo 'aaaaaaaaaaaaaaaaaaa';
    // \App\Models\User::factory(10)->create();
}

D:\lessons\first_project>php artisan migrate --seed
Nothing to migrate.
aaaaaaaaaaaaaaaaaaa Database seeding completed successfully.
```

Результат

```
public function run()
{
    // \App\Models\User::factory(10)->create();
}
```

Причем в методе идет обращение к классу factory

**ПОГОВОРИМ ДАЛЬШЕ ПОТОМ**

Factory (фабрика)

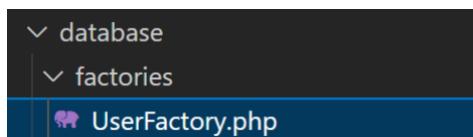
Это изначальная ФАБРИКА для пользователей, которую предоставляет ларавел

Для постов категорий и тегов нужно создать СВОИ ФАБРИКИ

**Основное назначение фабрики- дать готовые объекты на выходе**

**Объекты могут сразу отправиться в базу или просто стать массивом (и т.д.)**

В дальнейшем мы сможем работать с этими объектами



Расположение

Совместная работа

```
public function run()
{
$category = [];
$tag = [];
Category::create();
// \App\Models\User::factory(10)->create();
```

Можно все прописать в сиде НО НЕ НУЖНО ТК У НАС ЕСТЬ ФАБРИКА ФАБРИКА сделает все автоматически

```
php artisan make:factory PostFactory --model=Post
```

Создадим фабрику для постов с привязкой к модели (можно добавить равно)(не работает)

```
use App\Models\Post;
use Illuminate\Database\Eloquent\Factories\Factory;

class PostFactory extends Factory
{
    protected $model = Post::class;
```

Добавляем модель вручную

```
namespace Database\Factories;

use App\Models\Post;
use Illuminate\Database\Eloquent\Factories\Factory;

class PostFactory extends Factory
{
    protected $model = Post::class;
    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            //
        ];
    }
}
```

Содержание класса фактори

```
public function definition()
{
    return [
        "name" => "blabla",
        "old" => '19'
    ];
}
```

Этот метод при обращении к фабрики выдает массив

```
/*
public function run()
{
    \App\Models\User::factory(10)->create();
}
```

При вызове фабрики в сидре ОБРАЩЕНИЕ ИДЕТ К САМОЕ МОДЕЛИ

```
class Post extends Model
{
    use HasFactory;
```

Использование фабрики

```
namespace Database\Seeders;

use App\Models\Post;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $posts = Post::factory(10)->make();
        dd($posts);
        // \App\Models\User::factory(10)->create();
    }
}
```

Вызываем сид с фабрикой (не create а make чтобы ничего не создалось в бд а только вывелось на экран)

```
PS C:\OpenServer\domains\laravel\crud_relations> php artisan migrate --seed
Nothing to migrate.
Illuminate\Database\Eloquent\Collection^ {#961
    #items: array:10 [
        0 => App\Models\Post^ {#964
            #table: "posts"
            #guarded: []
            #connection: null
            #primaryKey: "id"
            #keyType: "int"
            +incrementing: true
            #with: []
            #withCount: []
            +preventsLazyLoading: false
            #perPage: 15
            +exists: false
            +wasRecentlyCreated: false
            #escapeWhenCastingToString: false
            #attributes: array:2 [
                "name" => "blabla"
                "old" => "19"
            ]
            #original: []
        }
    ]
}
```

Вывод на экран и вызов

Вернулись МОДЕЛИ с зарезервированными методами (В БАЗУ НЕ ПОШЛО ЭТО ПРОСТО МАССИВ С КЛАССОМ ПОСТ(поэтому есть зарезервированными методы))

```
public function definition()
{
    return [
        'title' => 'blabla',
        'content' => 'blablablabla',
        'image' => 'blablablabla',
        'likes' => random_int(1, 2000),
        'is_published' => 1,
        'category_id' => 1
    ];
}
```

Для того чтобы отправить созданный объект в бд у него должны быть те же атрибуты что и у таблицы (значения можно задавать рандомные!!)

```
public function run()
{
    $posts = Post::factory(10)->make();
    dd($posts);
    // \App\Models\User::factory(10)->create();
}
```

```
$posts = Post::factory(10)->create();
```

Создаем 10 объектов в бд

```
'category_id' => Category::get()->random()->id
```

Берем рандомную категорию

```
public function definition()
{
    return [
        'title' => $this->faker->title( gender: 20),
```

Рандомный заголовок!!!

```
D:\lessons\first_project>php artisan make:factory CategoryFactory
Factory created successfully.
```

```
D:\lessons\first_project>php artisan make:factory TagFactory
Factory created successfully.
```

Создадим фабрику для категорий и тегов

The screenshot shows the PHPStorm IDE interface. On the left is the project tree with files like Category.php, Post.php, Tag.php, User.php, CategoryFactory.php, PostFactory.php, TagFactory.php, and UserFactory.php. The TagFactory.php file is currently selected and open in the editor. The code defines a factory for the Tag model, setting its default state to a random word generated by Faker.

```
protected $model = Tag::class;

/**
 * Define the model's default state.
 */
public function definition()
{
    return [
        'title' => $this->faker->word
    ];
}
```

Заполняем

```
public function run()
{
    Category::factory( ...parameters: 20)->create();
    Post::factory( ...parameters: 200)->create();
    Tag::factory( ...parameters: 50)->create();
```

Создаем

```
public function run()
{
    Category::factory( ...parameters: 20 )->create();
    $tags = Tag::factory( ...parameters: 50 )->create();
    $posts = Post::factory( ...parameters: 200 )->create();

    foreach ( $posts as $post ) {
        $tagsIds = $tags->random( number: 5 )->pluck( value: 'id' );
        $post->tags()->attach( $tagsIds );
    }
}
```

Привязываем

Pluck выцепит только айди и вернет массив айдишников

## Пагинация в Laravel

Пагинация (Pagination) – это порядковая нумерация страниц, которая в основном размещается вверху либо внизу страниц сайта.

Пагинация ОЧЕНЬ сильно снижает нагрузку на сайт т. к. мы загружаем не сразу все 10000 постов на сайт, а по 10-20 на страницу

```
class IndexController extends BaseController
{
    public function __invoke()
    {
        $posts = Post::paginate(10);
        return view('post.index', compact('posts'));
    }
}
```

Меняем Post::all на вот это

- [1. impedit id nihil](#)
- [2. harum natus ducimus](#)
- [3. dolor qui voluptas](#)
- [4. debitis distinctio nobis](#)
- [5. repellendus ut at](#)
- [6. natus repellat rem](#)
- [7. rem ipsa ut](#)
- [8. sequi in doloremque](#)
- [9. sed nobis culpa](#)
- [10. architecto reiciendis aut](#)

Получаем

```
Illuminate\Database\Eloquent\Collection {#1424 ▾
  #items: array:200 [▶]
```

ЭТО МЫ ПОЛУЧАЕМ ПРИ ДД ВСЕХ ПОСТОВ

```
Illuminate\Pagination\LengthAwarePaginator {#1021 ▾
  #total: 200
  #lastPage: 20
  #items: Illuminate\Database\Eloquent\Collection {#1235 ▶}
  #perPage: 10
  #currentPage: 1
  #path: "http://127.0.0.1:8000/posts"
  #query: []
  #fragment: null
  #pageName: "page"
  +onEachSide: 3
  #options: array:2 [▶]
```

А ПРИ ПАГИНАЦИИ ВОТ ЭТО(через этот метод можно отобразить ссылки)

```
IndexController.php index.blade.php X
resources > views > post > index.blade.php > ...
1  @extends('layouts.post')
2  @section('content')
3    @foreach($posts as $post)
4      <a href="{{route('post.show', $post->id)}}">{{ $post->id }}. {{ $post->title }}</a>
5    @endforeach
6    <div>
7      {{$posts->links()}}
8    </div>
9  @endsection()
```

Добавляем ссылки

[11. iure iure quia](#)  
[12. error magnam vero](#)  
[13. repellat voluptatum et](#)  
[14. vel vitae vitae](#)  
[15. eveniet quam qui](#)  
[16. maiores totam nesciunt](#)  
[17. consequuntur doloribus qui](#)  
[18. accusantium omnis dicta](#)  
[19. optio tempora odit](#)  
[20. alias soluta accusamus](#)

[« Previous](#) [Next »](#)

Showing 11 to 20 of 200 results



Получаем вот что (выглядит все так некрасиво, потому что по умолчанию идет шаблон тейлвинд а не бутстррап)

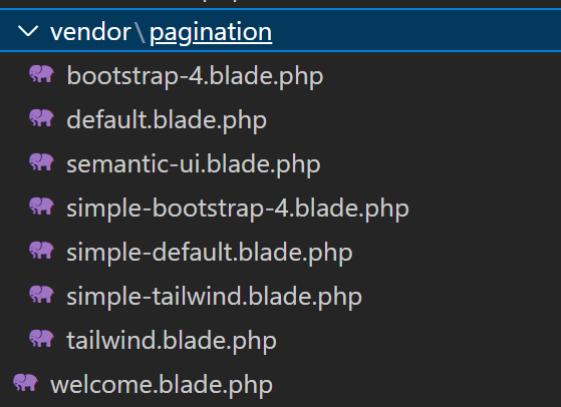
## Database: Pagination

Документация

```
PS C:\OpenServer\domains\laravel\crudapi> php artisan vendor:publish --tag=laravel-pagination
Copied Directory [\vendor\laravel\framework\src\Illuminate\Pagination\resources\views] To [\resources\views\ve
ndor\pagination]
Publishing complete.
```

Выполняем эту команду

Происходит линковка (копирование из одной папке в другую для удобства чтения)



Получаем шаблоны пагинации



Заходим в этот файл и меняем там шаблон

ВСЕ ЧИТАТЬ В ДОКУМЕНТАЦИИ

```
use Illuminate\Pagination\Paginator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Paginator::defaultView('vendor.pagination.bootstrap-4');
    }
}
```

Дефолтная пагинация

- [1. impedit id nihil](#)
- [2. harum natus ducimus](#)
- [3. dolor qui voluptas](#)
- [4. debitis distinctio nobis](#)
- [5. repellendus ut at](#)
- [6. natus repellat rem](#)
- [7. rem ipsa ut](#)
- [8. sequi in doloremque](#)
- [9. sed nobis culpa](#)
- [10. architecto reiciendis aut](#)

[«](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [...](#) [19](#) [20](#) [»](#)

Получаем

**ВСЕ ШАБЛОНЫ МОЖНО МЕНЯТЬ И ДОБАВЛЯТЬ НОВЫЕ**

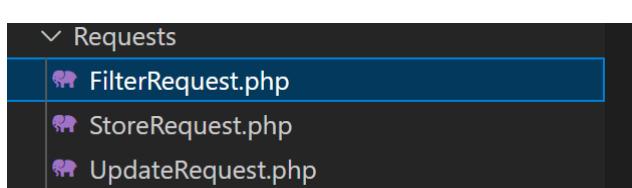
## Шаблон Filter, фильтрация данных в Laravel

### НЕИСПОЛЬЗУЕМЫЙ ШАБЛОН

```
class IndexController extends BaseController
{
    public function __invoke()
    {
        $posts = Post::where('is_published', 1)
            ->where('category_id', 1)
            ->get();
    }
}
```

Обычная фильтрация данных

Так делать не круто и это идиотизм т. к. нельзя ничего передать



Правильная фильтрация делается через реквесты т. к. они проверяют данные

```
i 127.0.0.1:8000/posts?category_id=5
```

Это называется query string или строка запроса (гет запрос после ?)

```
class IndexController extends BaseController
{
    public function __invoke(FilterRequest $request){
        $data = $request->validated();

        $query = Post::query();
        dd($query);

        if(isset($data['category_id'])){
            $query->where('category_id', $data['category_id']);
        }

        $post = $query->get();

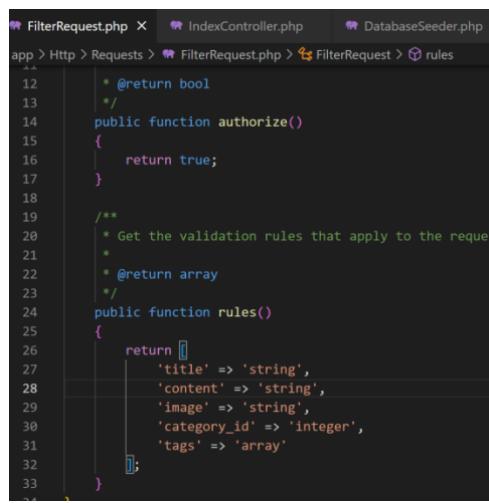
        dd($post);
    }
}
```

Так по получаем наши данные из реквеста и валидируем их

Далее конструируем query динамичный запрос

Проверяем условия и строим запрос

Выводим результат



```
FilterRequest.php X IndexController.php DatabaseSeeder.php
app > Http > Requests > FilterRequest.php > FilterRequest > rules
12     * @return bool
13     */
14    public function authorize()
15    {
16        return true;
17    }
18
19    /**
20     * Get the validation rules that apply to the request
21     *
22     * @return array
23     */
24    public function rules()
25    {
26        return [
27            'title' => 'string',
28            'content' => 'string',
29            'image' => 'string',
30            'category_id' => 'integer',
31            'tags' => 'array'
32        ];
33    }

```

Фильтр

```
Illuminate\Database\Eloquent\Collection {#351 ▾
  #items: array:27 [▼
    0 => App\Mode_\Post {#352 ▶}
    1 => App\Mode_\Post {#353 ▶}
    2 => App\Mode_\Post {#354 ▶}
    3 => App\Mode_\Post {#355 ▶}
    4 => App\Mode_\Post {#356 ▶}
    5 => App\Mode_\Post {#357 ▶}
    6 => App\Mode_\Post {#358 ▶}
    7 => App\Mode_\Post {#359 ▶}
    8 => App\Mode_\Post {#360 ▶}
    9 => App\Mode_\Post {#361 ▶}
    10 => App\Mode_\Post {#362 ▶}
    11 => App\Mode_\Post {#363 ▶}
    12 => App\Mode_\Post {#364 ▶}
    13 => App\Mode_\Post {#365 ▶}
    14 => App\Mode_\Post {#366 ▶}
    15 => App\Mode_\Post {#367 ▶}
    16 => App\Mode_\Post {#368 ▶}
    17 => App\Mode_\Post {#369 ▶}
    18 => App\Mode_\Post {#370 ▶}
    19 => App\Mode_\Post {#371 ▶}
    20 => App\Mode_\Post {#372 ▶}
    21 => App\Mode_\Post {#373 ▶}
    22 => App\Mode_\Post {#374 ▶}
    23 => App\Mode_\Post {#375 ▶}
    24 => App\Mode_\Post {#376 ▶}
    25 => App\Mode_\Post {#377 ▶}
    26 => App\Mode_\Post {#378 ▶}
  ]
  #escapeWhenCastingToString: false
}
```

Получаем

```
if(isset($data['title'])){
|   $query->where('title', 'like', "%{$data['title']}%");
}
```

Такая конструкция будет искать не только по полному слову но и при любом сочетании букв введенных букв в любом месте предложения

```
class IndexController extends BaseController
{
    public function __invoke(FilterRequest $request){
        $data = $request->validated();

        $query = Post::query();
        // dd($query);

        if(isset($data['category_id'])){
            $query->where('category_id', $data['category_id']);
        }

        if(isset($data['title'])){
            $query->where('title', 'like', "%{$data['title']}%");
        }

        $post = $query->get('title');

        dd($post);
        $posts = Post::paginate(10);
        return view('post.index', compact('posts'));
    }
}
```

Если по ключу запросы не приходят, то условия просто игнорируются

Можно делать вот так

### Используемый шаблон

Шаблон выше не эффективен т. к. уже давно есть заранее придуманный шаблон

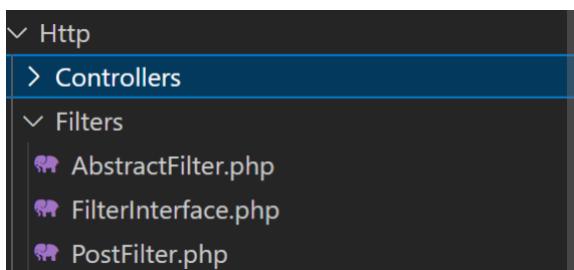
<https://www.youtube.com/watch?v=cL1eXKsnRJI&t=163s> ВИДЕО О ФИЛЬТРАХ

ЭТОТ ШАБЛОН СЛОЖНЫЙ  
УЧИТЬ ЕГО НЕ НАДО

НУЖНО ПРОСТО ЗАПОМНИТЬ И ПОНЯТЬ, ЧТО КАК СТОИТ

ИЛИ ТУПО КОПИРОВАТЬ

Данный шаблон лишь один из реализаций фильтров

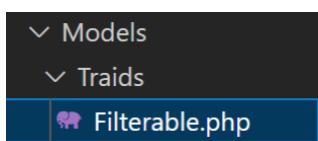


Создаем вот такой шаблон (2 класса 1 интерфейс) (два основных и один дополнительный)

Далее мы заполняем два обязательных файла

ФАЙЛЫ С КОММЕНТИРИЕМИ В КОНЦЕ ДОКУМЕНТАЦИИ (ФИЛЬТРЫ)

Далее в моделях мы должны создать трейды(контракты)



Название по конвенции (able - для всех трейдов)

Контракты в Laravel — это набор интерфейсов, которые описывают основной функционал, предоставляемый фреймворком. Например, контракт `Illuminate\Contracts\Queue\Queue` определяет методы, необходимые для организации

очередей, в то время как контракт `lluminate\Contracts\Mail\Mailer` определяет методы, необходимые для отправки электронной почты.

Трейды подключаются к моделям или класса для дополнения их функционала

```
class Post extends Model
{
    use HasFactory;
    use Filterable;
    use SoftDeletes;
```

Далее нужно создать фильтр на основе класса в контроллере

```
class IndexController extends BaseController
{
    public function __invoke(FilterRequest $request){
        $data = $request->validated();

        $filter = app()->make(PostFilter::class, ['queryParams'=>array_filter($data)]);

        $post = Post::filter($filter);
        dd($post);

        $posts = Post::paginate(10);
        return view('post.index', compact('posts'));
    }
}

$post = Post::filter($filter)->paginate(10);
```

ТАК ДЕЛАЕТСЯ ПАГИНАЦИЯ

SaintAdmin    Создать пост

[1. voluptatum architecto autem](#)  
[5. et molestiae autem](#)  
[13. fuga aut aut](#)  
[19. aut quod velit](#)  
[64. natus aut et](#)  
[82. nobis aut beatae](#)  
[93. ullam et autem](#)  
[95. sed aut id](#)  
[101. aut ut voluptatem](#)  
[114. aut rem autem](#)

« 1 2 »

## Результат

```
@extends('layouts.main')
@section('content')
    <div>
        <div>
            <a href="{{ route('post.create') }}" class="btn btn-primary mb-3">Add one</a>
        </div>

        @foreach($posts as $post)
            <div><a href="{{ route('post.show', $post->id) }}>{{ $post->id }}. {{ $post->title }}</a>
        @endforeach

        <div class="mt-3">
            {{ $posts->withQueryString()->links() }}
        </div>
    </div>
@endsection
```

Чтобы при переходе по страницам запросы не пропадали



127.0.0.1:8000/posts?title=ut&page=2

НА ЭТОМ ВСЕ

УЧИТЬ ЭТО НЕ НАДО

НУЖНО ПРОСТО ПОНЯТЬ, ЧТО КАК РАБОТАЕТ И ОРИЕНТИРОВАТЬСЯ

## Admin LTE в Laravel, устанавливаем админку

Админки устанавливаются +- одинаково

Мы возьмём админ лте тк она простая и популярная (можно использовать в проекте)

Скачиваем adminLte и распаковываем в папку

[https://www.youtube.com/watch?v=w9kUQZDAWas&list=PLd2\\_Os8Cj3t8pnG4ubQemoanTwf0VFEtU&index=32](https://www.youtube.com/watch?v=w9kUQZDAWas&list=PLd2_Os8Cj3t8pnG4ubQemoanTwf0VFEtU&index=32)

Смотреть видос или прошлые проекты

Спасибо за урок, чтобы не менять в шаблоне все пути, просто в head можно прописать, например под <title></title> вот такой код: <base href="{{ asset('/')}}">

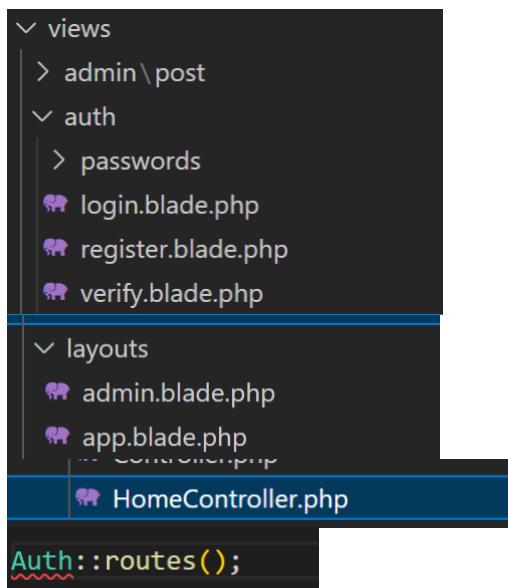
Я установил AdminLTE через композер. Команда для установки на их сайте в разделе Docs / Installation: после установки не нужно проставлять нигде {{ assets() }}, все прекрасно работает)

## Авторизация Laravel

В ларавел есть нативные средства авторизации

```
PS C:\OpenServer\domains\laravel\laravel> php artisan ui:auth
Authentication scaffolding generated successfully.
```

Подключение



Подключаемые файлы

```
Route::get('/', 'HomeController@index');
```

Добавляем на главную страницу homecontroller

Получаем

MySQL вернула пустой результат (т.е. ноль строк). (Запрос занял 0,0004 сек.)

```
SELECT * FROM `users`
```

id name email email\_verified\_at password remember\_token created\_at updated\_at

Использование результатов запроса

Автоматом уже заранее есть база данных с пользователями

□	1	Artem	muzychukartem.music@gmail.com	NULL	\$2y\$10\$NNWOYmJTwPgtJGUi0IBkZes1iLWGgFriHUu6ndNVcfP...	

ВСЕ ЭТО ЛЕГКО РЕДАКТИРУЕТСЯ В ИСХОДНЫХ ФАЙЛЫХ

```
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => ['required', 'string', 'max:255'],
        'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],
        'password' => ['required', 'string', 'min:8', 'confirmed'],
    ]);
}
```

НАЗВАНИЯ ДОЛЖНЫ БЫТЬ ПРАВИЛЬНЫМИ И ПОСМОТРЕТЬ ИХ МОЖНО ТУТ

{auth()->user()->name}

Поменять <https://stackoverflow.com/questions/73180945/jetstream-css-and-js-not-working-and-showing-viteresources-css-app-css-re>

## Класс Middleware в Laravel (Роли авторизации)

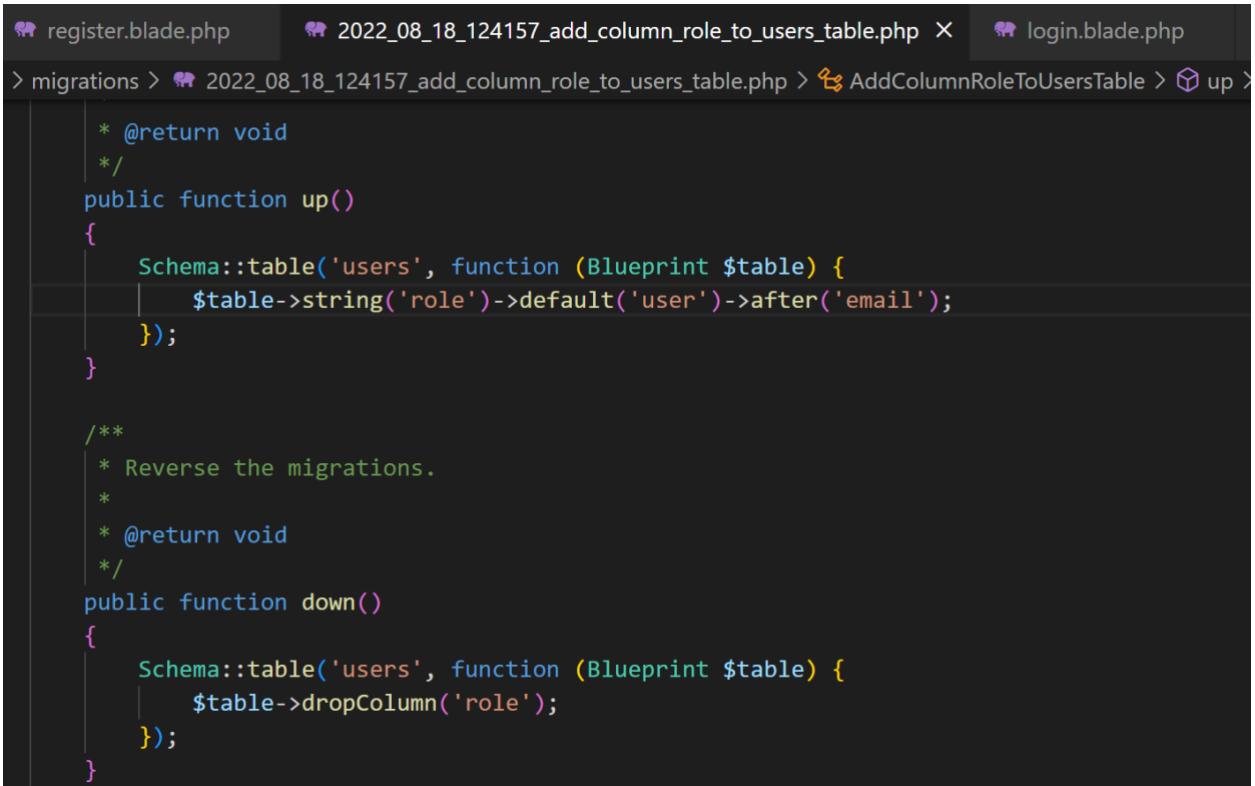
Пока что наша регистрация просто ни о чём

Правильная регистрация осуществляется с распределением ролей чтобы простой юзер не попал в админку

Middleware - в переводе - ПОСРЕДНИК(что-то между) МЕЖДУ ЗАПРОСОМ И КОНТРОЛЛЕРОМ

Его прямая обязанность допускать работу в контроллере или не допускать (ПРОВЕРЯТЬ РОЛЬ)

### 1 Меняем миграцию users



The screenshot shows a code editor with several tabs at the top: 'register.blade.php', '2022\_08\_18\_124157\_add\_column\_role\_to\_users\_table.php', and 'login.blade.php'. The main content area displays a PHP migration file for the 'users' table:

```
* @return void
*/
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        $table->string('role')->default('user')->after('email');
    });
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('role');
    });
}
```

### 2 Создаем посредника

```
> php artisan make:middleware AdminPanelMiddleware
```

The screenshot shows a file browser window with a tree view. Under the 'Middleware' folder, several files are listed: AdminPanelMiddleware.php, Authenticate.php, EncryptCookies.php, PreventRequestsDuringMaintenance.php, RedirectIfAuthenticated.php, TrimStrings.php, TrustHosts.php, TrustProxies.php, and VerifyCsrfToken.php. Below this, the content of AdminPanelMiddleware.php is displayed in a code editor:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class AdminPanelMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param Closure(\Illuminate\Http\Request): (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse) $next
     */
    public function handle(Request $request, Closure $next)
    {
        return $next($request);
    }
}
```

Содержимое посредника

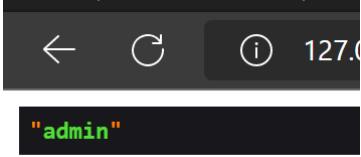
В нем у нас происходит некая проверка и по ее завершению мы что-то выполняем или пересылаем при запрете

```
$user = User::where('is_auth', 1)
auth()->
```

Данный хелпер вытягивает пользователя, который заходит на сайт ИМЕННО НАС НА НАШЕМ КОМПЕ

```
public function handle(Request $request, Closure $next)
{
    dd(auth()->user()->role);
    return $next($request);
}
```

Так мы можем вытянуть роль



```
public function handle(Request $request, Closure $next)
{
    if(auth()->user()->role !== 'admin'){
        return response('You are not an admin');
    }
    return $next($request);
}
```

Делаем проверку

### 3 Регистрируем посредника



Сюда

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'admin' => AdminPanelMiddleware::class
];
```

Тут

### 3 Прописываем атрибуты в роутах

```
Route::group(['namespace' => 'Admin', 'prefix' => 'admin', 'middleware' => 'admin'], function(){
    Route::group(['namespace' => 'Post'], function(){
        Route::get('/post', 'IndexController')->name('admin.post.index');
    });
});
```

Теперь всегда при переходе в админку будет проходить проверка на админа

## Класс Policy

**Middleware** - разрешение и запрет доступа

**Policy** - демонстрация и скрытие доступа (ссылки, контент, функции и т.д) - СКРЫТЬ ЭЛЕМЕНТЫ ИСХОДЯ ИЗ РОЛИ ПОЛЬЗОВАТЕЛЯ

Например, чтобы постоянно не набирать в строке запроса админ, а чтобы у админа уже была кнопка перехода

```
<nav>
    <li class="nav-item">
        <a href="{{route('admin.post.index')}}">АдминПанель</a>
    </li>
</nav>
```

Создаем какой-нибудь элемент, который должен видеть только админ

```
php artisan make:policy AdminPolicy -m User
```

Создаем СЛЕДОВАТЕЛЯ с привязкой к модели пользователь

```
/*
public function view(User $user, User $model)
{
    //
}
```

Все функции следователя относятся к CRUD

```
public function forceDelete(User $user, User $model)
{
    //
}
```

Удаляет пост с учетом софт делит (полностью удалит)

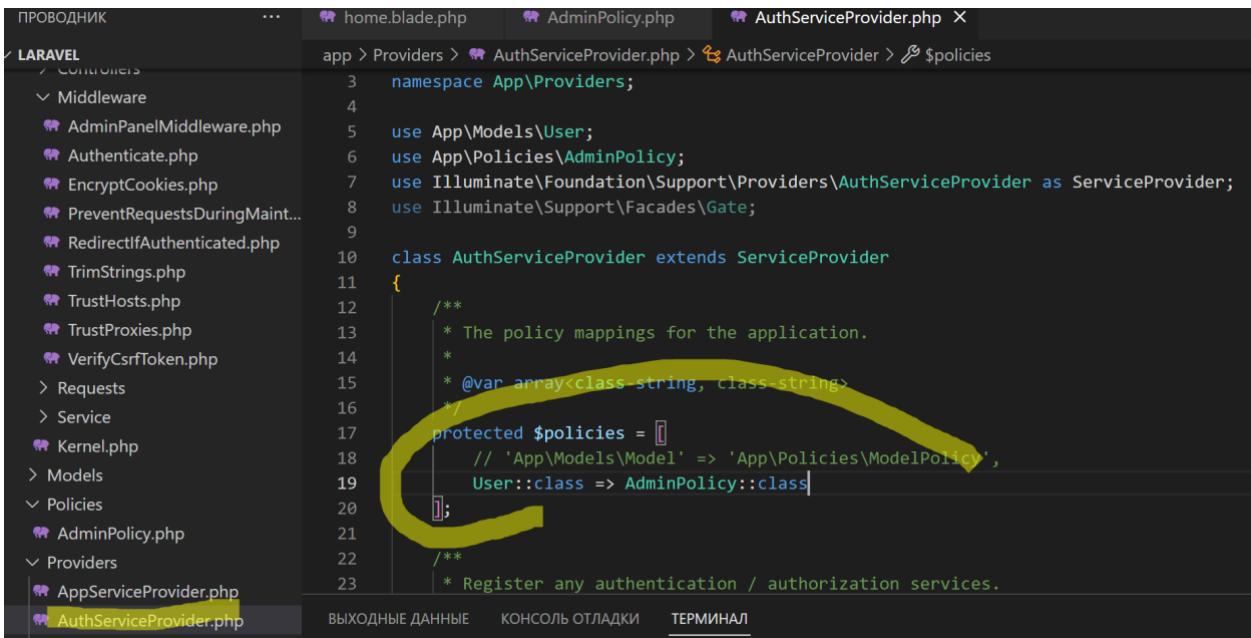
```
public function view(User $user, User $model)
{
    if($model->role !== 'admin'){
        return false;
    }
}
```

Отображение и скрытие элементов происходит через метод view

```
public function view(User $user, User $model)
{
    return $model->role === 'admin';
}
```

Можно так т.к. метод уже делает проверку

**СЛЕДОВАТЕЛЬ МОЖЕТ ИСПОЛЬЗОВАТЬСЯ В КОНТРОЛЛЕРАХ И ВЬЮВАХ**



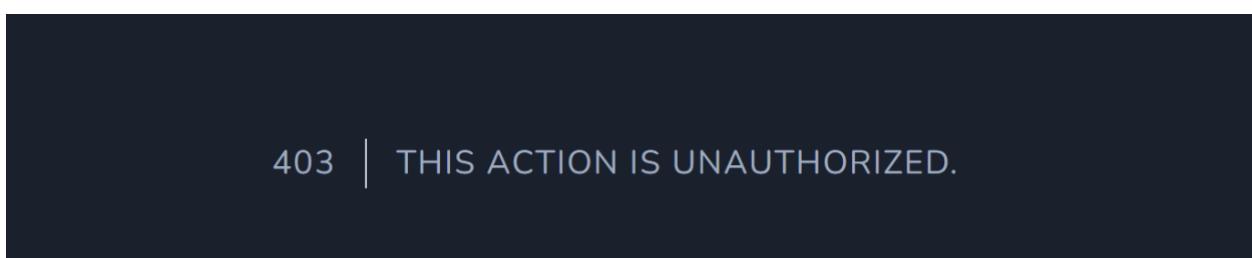
```
ПРОВОДНИК ... home.blade.php AdminPolicy.php AuthServiceProvider.php X
LARAVEL Controllers
    Middleware
        AdminPanelMiddleware.php
        Authenticate.php
        EncryptCookies.php
        PreventRequestsDuringMaint...
        RedirectIfAuthenticated.php
        TrimStrings.php
        TrustHosts.php
        TrustProxies.php
        VerifyCsrfToken.php
    Requests
    Service
        Kernel.php
    Models
    Policies
        AdminPolicy.php
    Providers
        AppServiceProvider.php
        AuthServiceProvider.php
```

```
app > Providers > AuthServiceProvider.php > AuthServiceProvider > $policies
3     namespace App\Providers;
4
5     use App\Models\User;
6     use App\Policies\AdminPolicy;
7     use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
8     use Illuminate\Support\Facades\Gate;
9
10    class AuthServiceProvider extends ServiceProvider
11    {
12        /**
13         * The policy mappings for the application.
14         *
15         * @var array<class-string, class-string>
16         */
17        protected $policies = [
18            // 'App\Models\Model' => 'App\Policies\ModelPolicy',
19            User::class => AdminPolicy::class
20        ];
21
22        /**
23         * Register any authentication / authorization services.
```

Регистрируем следователя

```
class IndexController extends BaseController
{
    public function __invoke(){
        $this->authorize('view', auth()->user());
        $posts = Post::paginate(10);
        $count = 1;
        return view('post.index',compact('posts','count'));
    }
}
```

Делаем в контроллере проверку на роль



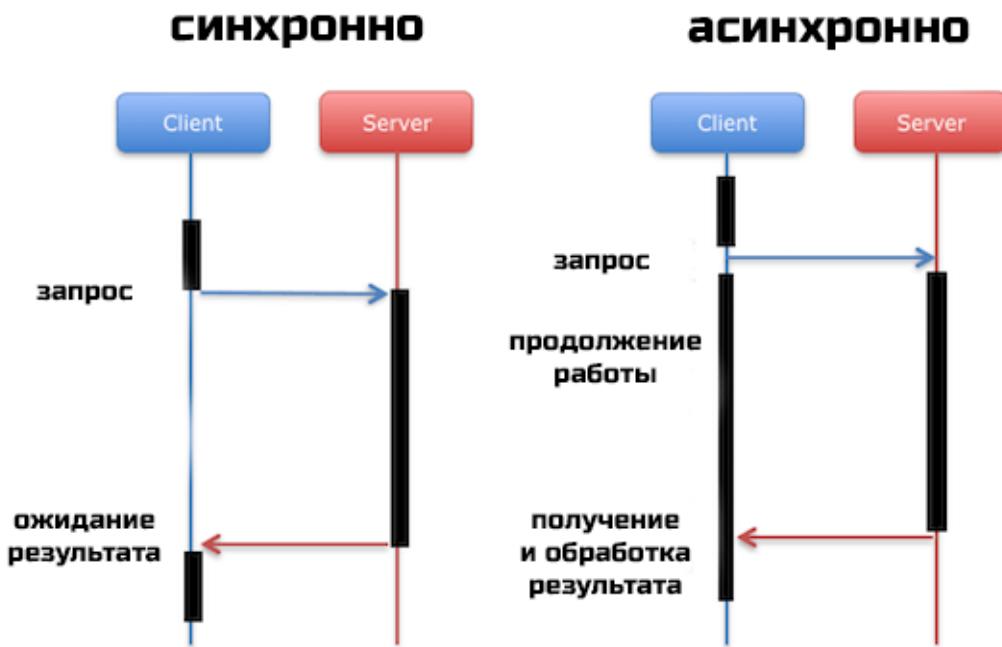
Ошибка если роль не совпадает с админом

```
@can('view',auth()->user())
<nav>
    <li class="nav-item">
        <a href="{{route('admin.post.index')}}">АдминПанель</a>
    </li>
</nav>
@endcan
```

ТАК МОЖНО СКРЫТЬ ЭЛЕМЕНТЫ

## Асинхронный CRUD в Laravel. Приложение Postman.

Postman – приложение, имитирующее запросы с фронтенда  
аjax – технология работы загрузчика и обновление информации на сайте без перезагрузки сайта) работает это благодаря АСИНХРОННЫМ ЗАПРОСАМ



<https://russianblogs.com/article/7728163812/>  
<https://www.youtube.com/watch?v=3D2kYmEa8rk>

Постман используется для теста запросов перед продакшином (реакция ответа на асинхронный запрос с сайта) (имитация запросов с фронта)

The screenshot shows the Postman application interface. On the left, there's a sidebar with various sections like Collections, APIs, Environments, Mock Servers, Monitors, Flows, and History. A central area displays a collection named 'SAINT' containing a folder 'My first collection' with two requests: 'First folder inside collection' (POST) and 'Second folder inside collection' (GET). Below this, there's a 'Create a collection for your requests' section with a 'Create collection' button. To the right, a new request panel is open for 'Untitled Request' with a 'GET' method and an 'Enter request URL' field. Below the URL field are tabs for Params, Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings. A 'Query Params' table is also present. At the bottom of the interface, there are buttons for Save, Send, and a list of recent items.

## Интерфейс

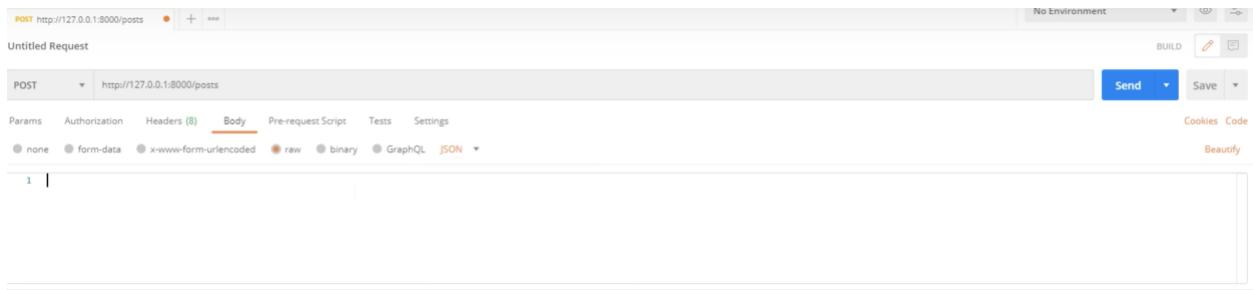
The screenshot shows a single request in the Postman interface. The method is 'GET' and the URL is 'http://127.0.0.1:8000/posts'. The 'Send' button is visible to the right. Below the request, the text 'Сюда можно вводить наши запросы' (Here you can enter our requests) is displayed. The response tab is selected, showing the status '200 OK' and '79 ms' latency. The response body is a rendered HTML page with the title 'SaintPosts' and a navigation bar. The HTML code is shown in the 'Pretty' tab:

```

1 <html lang="en">
2
3   <head>
4     <meta charset="UTF-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <link rel="stylesheet" href="http://127.0.0.1:8000/css/app.css">
8     <title>SaintPosts</title>
9   </head>
10
11  <body>
12    <div class="container">
13      <header class="row">
14        <nav class="navbar navbar-expand-lg navbar-light bg-light">
15          <div class="container-fluid">
16            <a class="navbar-brand" href="http://127.0.0.1:8000/posts">Saint-Posts</a>
17            <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
18                  data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false"
19                  aria-label="Toggle navigation">
20              <span class="navbar-toggler-icon"></span>
21            </button>
22            <div class="collapse navbar-collapse" id="navbarNav">

```

Ниже получим ответ (кнопки выше меняют вид)



Так мы можем создавать что-то в базе для тестов (body-row-json)

Json тут используется как ассоциативный массив

```
1 {  
2   ... "name": "boris",  
3   ... "age": 20  
4 }
```

Синтаксис

```
return [  
    'title' => 'string',  
    'content' => 'string',  
    'image' => 'string',  
    'category_id' => '',  
    'tags' => '',  
];|
```

Наше поле ожидает вот такие ключи

НО РАБОАТЬ НИЧЕГО НЕ БУДЕТ ТК НУЖЕН csrf токен  
ДЛЯ ЭТОГО МЫ ДОБАВИМ НУЖНЫЙ РОУТ В ИСКЛЮЧЕНИЯ

```
VerifyCsrfToken.php x StoreRequest.php web.php  
laravel > laravel > app > Http > Middleware > VerifyCsrfToken.php  
6  
7 class VerifyCsrfToken extends Middleware  
8 {  
9     /**  
10      * The URIs that should be excluded from CSRF verification.  
11      *  
12      * @var array<int, string>  
13      */  
14     protected $except = [  
15         '/posts/store'  
16     ];  
17 }
```

СЮДА (в except)

```

1
2   "title": "boris",
3   "content": "20",
4   "images": 2,
5   "status": 0,
6   "category_id": 6
7

```

```

1
2   "title": "artem",
3   "content": "123",
4   "images": "123",
5   "status": 1,
6   "category_id": 5
7

```

## Запрос в постман

Постман помогает нам отслеживать ошибки, тестировать запросы и смотреть программную часть сайта

## Класс Resource в Laravel, асинхронный ответ с бека. Restful API

ВСЕ ЧТО БУДЕТ НИЖЕ ЭТО restful api

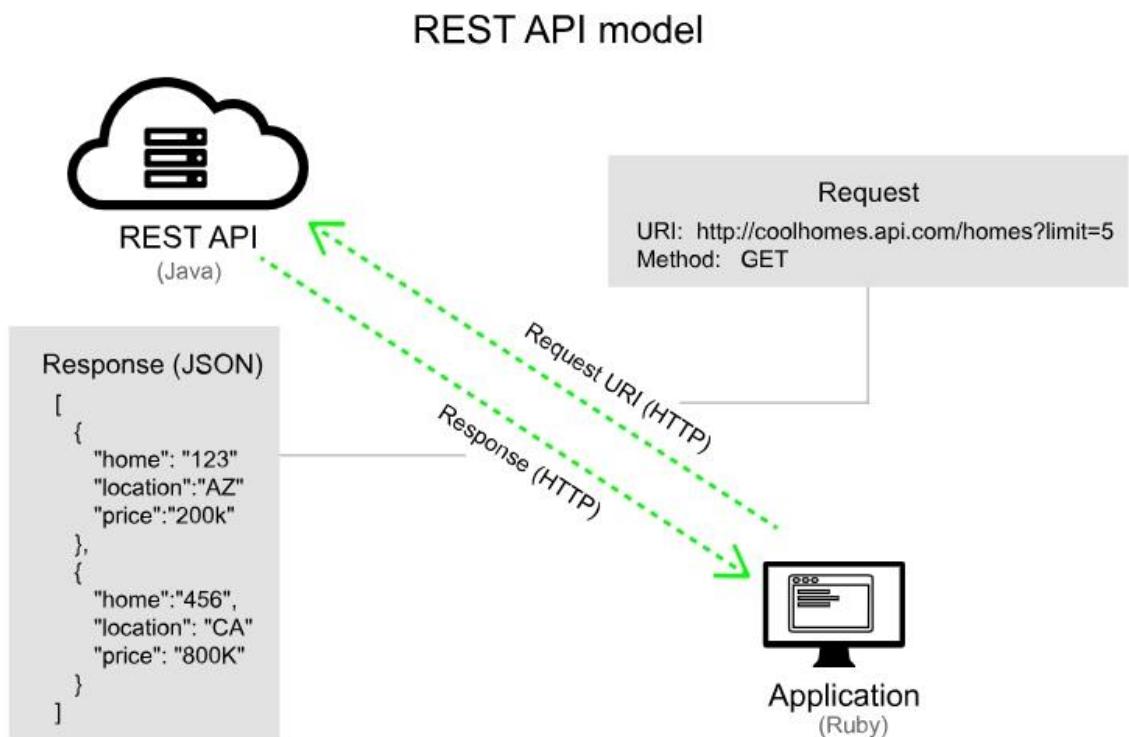
<https://aws.amazon.com/ru/what-is/restful-api/>

RESTful API — это интерфейс, используемый двумя компьютерными системами для безопасного обмена информацией через Интернет.

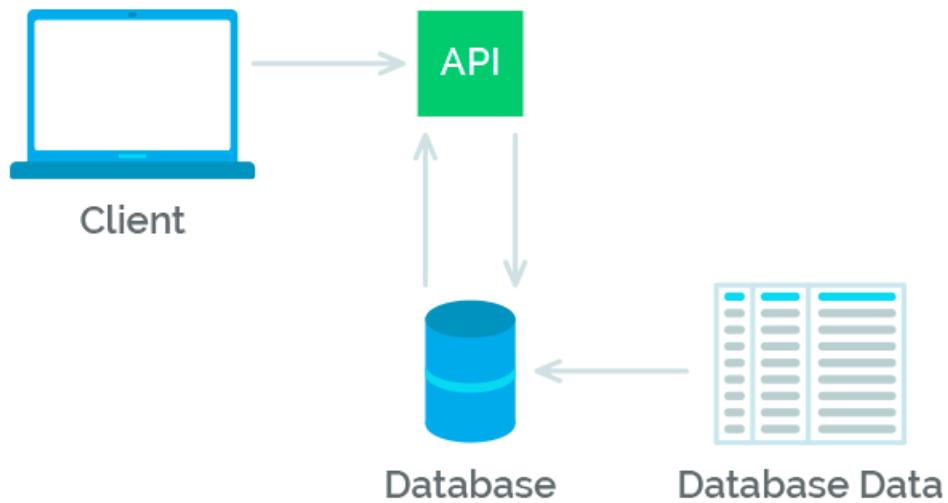
Representational State Transfer (REST) — это программная архитектура, которая определяет условия работы API. Первоначально REST создавалась как руководство для управления взаимодействиями в сложной сети, такой как Интернет.

Большинство бизнес-приложений должны взаимодействовать с другими внутренними и сторонними приложениями для выполнения различных задач. Например, чтобы генерировать ежемесячные платежные ведомости, ваша внутренняя бухгалтерская система должна обмениваться данными с банковской системой вашего клиента, чтобы автоматизировать выставление счетов и взаимодействовать с внутренним приложением по учету рабочего времени. RESTful API поддерживают такой обмен информацией, поскольку они следуют безопасным, надежным и эффективным стандартам программного взаимодействия.

Веб-службы, реализующие архитектуру REST, называются веб-службами RESTful. Как правило, термин RESTful API относится к сетевым RESTful API. Однако REST API и RESTful API являются взаимозаменяемыми терминами.



## REST API Design



принципы архитектурного стиля REST:

Единый интерфейс  
отсутствие сохранения состояния  
многоуровневой системной  
кэширование  
Код по запросу

REST API не зависит от используемой технологии. Вы можете создавать как клиентские, так и серверные приложения на разных языках программирования, не затрагивая структуру API. Также можно изменить базовую технологию на любой стороне, не влияя на обмен данными.

Как работает RESTful API?

Базовый принцип работы RESTful API совпадает с принципом работы в Интернете. Клиент связывается с сервером с помощью API, когда ему требуется какой-либо ресурс. Разработчики описывают принцип использования REST API клиентом в документации на API серверного приложения. Ниже представлены основные этапы запроса REST API:

1. Клиент отправляет запрос на сервер. Руководствуясь документацией API, клиент форматирует запрос таким образом, чтобы его понимал сервер.
2. Сервер аутентифицирует клиента и подтверждает, что клиент имеет право сделать этот запрос.
3. Сервер получает запрос и внутренне обрабатывает его.
4. Сервер возвращает ответ клиенту. Ответ содержит информацию, которая сообщает клиенту, был ли запрос успешным. Также запрос включает сведения, запрошенные клиентом.

## HTTP-аутентификация

HTTP определяет некоторые схемы аутентификации, которые можно использовать при реализации REST API. Ниже представлены две такие схемы:

### *Базовая аутентификация*

При базовой аутентификации клиент отправляет имя пользователя и пароль в заголовке запроса. Он кодирует их с помощью метода кодирования base64, который преобразует пару имя пользователя–пароль в набор из 64 символов для безопасной передачи.

### *Аутентификация носителя*

Аутентификация носителя — это процесс предоставления управления доступом носителю токена. Как правило, токен носителя представляет собой зашифрованную строку символов, которую сервер генерирует в ответ на запрос входа в систему. Клиент отправляет токен в заголовках запроса для доступа к ресурсам.

## OAuth

OAuth сочетает в себе пароли и токены для безопасного входа в любую систему. Сначала сервер запрашивает пароль, а затем дополнительный токен для завершения процесса авторизации. Он может проверять токен в любое время, а также через определенный период времени в соответствии с областью и сроком действия.

Асинхронный ответ в бека на асинхронный запрос должен быть json формата

Асинхронный ответ с бэка должен быть в виде массивы json формата для удобной работы

Вот как это сделать:

Для этого в ларавел есть класс Resource

```
namespace App\Services\Post;

use App\Models\Post;

class Service
{
    public function store($data)
    {
        $tags = $data['tags'];
        unset($data['tags']);

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

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

        return $post;
    }
}
```

СМ на ретурн

```
class StoreController extends BaseController
{
    public function __invoke(StoreRequest $request)
    {
        $data = $request->validated();

        $post = $this->service->store($data);

        $arr = [
            'title' => $post->title,
            'content' => $post->content,
            'image' => $post->image,
        ];
        return $arr;
        return redirect()->route('post.index');
    }
}
```

Body Cookies (2) Headers (9) Test Results

Pretty

Raw

Preview

Visualize

JSON



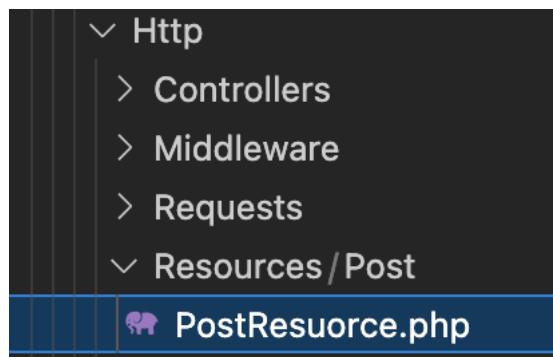
```
1  {
2      "title": "Boris",
3      "content": "Some content",
4      "image": "blabla.jpeg"
5 }
```

Вообще можно сделать все вот таким способом чтобы получить json ответ от бека, но в ларавел есть свой класс чтобы не писать все вот это

Resource – по сути просто сформированный массив данных которые потом преобраз в json контент

```
php artisan make:resource Post/PostResource
```

Создаем ресурс



Расположение

```
<?php

namespace App\Http\Resources\Post;

use Illuminate\Http\Resources\Json\JsonResource;

class PostResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request $request
     * @return array|\Illuminate\Contracts\Support\Arrayable\JsonSerializable
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }
}
```

Содержание

Теперь создадим json ответ на асинхронный запрос с фронта при помощи ресурса(класса, который форматирует все json массив)

```

public function toArray($request)
{
    return [
        "title" => $this->title,
        "content" => $this->content,
    ];
}

```

Меняем в ресурсе (пост попадает в виде контекста (this = пост))

```

public function __invoke(StoreRequest $request){
    $data = $request -> validated();
    $post = $this->service->store($data);

    return new PostResource($post);
    // return redirect() -> route('post.index');
}

```

Меняем в сторконтроллере

По сути, мы просто сформировали ответ через ресурс

<http://127.0.0.1:8000/posts/store>

The screenshot shows the Postman application interface. A POST request is being made to <http://127.0.0.1:8000/posts/store>. The 'Body' tab is active, displaying a JSON object:

```

1 {
2     "title": "boris",
3     "content": "20",
4     "images": "2",
5     "status": 0,
6     "category_id": 6
7 }

```

The response tab shows the following details:

- Status: 201 Created
- Time: 74 ms
- Size: 1.14 KB
- Save Response

The response body is also displayed in a pretty-printed JSON format:

```

1 {
2     "data": {
3         "title": "boris",
4         "content": "20"
5     }
6 }

```

Отправляем асинхронный запрос и получаем ответ в виде json

Data – общепринятый стандарт при возвращении ресурса (ресурс возвращается с ключом data)

Это для 1 поста. А если их 10?

```
return PostResource::collection($posts);
```

В indexcontroller теперь не создаем новый класс ресурса, а обращаемся к его статистическому методу (если мы знаем, что ответ будет не один пост) (без new тк будет не один пост, а массив постов)

PostResource прежде всего просто php класс

The screenshot shows the Postman interface with a GET request to `http://127.0.0.1:8000/posts`. The response body is a JSON array containing 10 posts, each with a title and content. The JSON is displayed in a pretty-printed format.

```
1 {
2   "data": [
3     {
4       "title": "delectus incident animi",
5       "content": "Qui quas repellendus maiores est. Tenetur nam sint error eligendi. Labore recusandae placeat est."
6     },
7     {
8       "title": "ut consequatur officia",
9       "content": "Nobis id sint fuga ea. Commodi dolor perferendis non qui. Vero itaque aliquid non."
10    },
11    {
12      "title": "eveniet neque",
13      "content": "Reprehenderit facilis similique rerum. Repudiandae vitae perferendis porro."
14    }
]
```

Обращаемся через гет и видим резу

**ФИЛЬТРАЦИЯ:**

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```

1 {
2   ... "title": "ut"
3 }

```

Body Cookies (2) Headers (9) Test Results 200 OK 60 ms 3.5s

Pretty Raw Preview Visualize JSON

```

1 {
2   "data": [
3     {

```

В БОДИ -РОУ ФОРМИРУЕМ ФИЛЬТР

При подключении фильтра в этом поле можно фильтровать запросы

## ПЕРЕЛИСТЫВАНИЕ СТРАНИЦ

```

DestroyController.php
EditController.php
IndexController.php
ShowController.php
StoreController.php
UpdateController.php
AboutController.php
ContactController.php
Controller.php
HomeController.php
MainController.php
MyPlaceController.php
PostController.php
> Filters
> Middleware
> Requests
  > Post
    FilterRequest.php
    StoreRequest.php
    UpdateRequest.php
> Resources
  Kernel.php
Models
Policies
Providers
Services

```

```

23 */
24 public function rules()
25 {
26   return [
27     'title' => '',
28     'content' => '',
29     'category_id' => '',
30     'page' => '',
31     'per_page' => |
32   ];
33 }
34 }
35 }
36

```

Для реализации перелистывания фильтра на странице постмана нужно добавить в фильтр две св-ва(последние два)

```
public function __invoke(FilterRequest $request)
{
    $data = $request->validated();

    $page = $data['page'] ?? 1;
    $perPage = $data['per_page'] ?? 10;

    $filter = app()->make(AbstractPostFilter::class, ['queryParams' => array_filter($data)]);
    $posts = Post::filter($filter)->paginate($perPage, ['*'], 'page', $page);

    return PostResource::collection($posts);
}

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

### Меняем индекс контроллер

```
GET http://127.0.0.1:8000/posts

Params Authorization Headers (9) Body Pre-request Script Test
none form-data x-www-form-urlencoded raw binary Gr

1 {
2     "title": "aut",
3     "page": 2
4 }

Body Cookies (2) Headers (9) Test Results
Pretty Raw Preview Visualize JSON ▾

1 {
2     "data": [
3         {
4             "id": 85,
5             "title": "Ad laudantium aut iusto alias unde.",
6             "content": "Tempore ullam eveniet enim aperiam exercitatio",
7             "image": "https://via.placeholder.com/640x480.png/0088aa?#"
8         },
9     {

```

Теперь при запросе мы можем конкретно указать какую стр отобразить  
Пока не надо см видео 36

Апдейт работает так же

```
        return $post;
    }

    public function update($post, $data)
    {
        $tags = $data['tags'];
        unset($data['tags']);

        $post->update($data);
        $post->tags()->sync($tags);
        $post = $post->fresh()
    }
}
```

Меняем сервис (последняя строка) (принудительное обновление базы)

```
class UpdateController extends BaseController
{
    public function __invoke(UpdateRequest $request, Post $post)
    {
        $data = $request->validated();

        $post = $this->service->update($post, $data);

        return new PostResource($post);
    }
}
```

Контроллер

```
use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        '/posts',
        '/posts/*',
    ];
}
```

## Убрали защиту

The screenshot shows the Postman interface. A request is being made to `PATCH http://127.0.0.1:8000/posts/7`. The `Body` tab is selected, showing a JSON payload with `"title": "112"`. The response status is `200 OK`, and the response body is displayed as:

```
1
2   "data": {
3     "id": 7,
4     "title": "112",
5     "content": "Velit et voluptatem rem omnis. Sed ut sapiente omnis maiores. Et dolorem sit et dolor dolor."
6   }
7 }
```

## JWT Token. Асинхронные роуты

`composer require php-open-source-saver/jwt-auth`

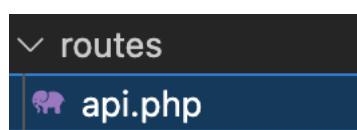
**ОБЯЗАТЕЛЬНО ПОЧИТАТЬ И ПОСОМТРЕТЬ ПРО jwt ТЕХНОЛОГИЯ КРУТАЯ И ВАЖНАЯ**

На самом деле роуты в `web.php` они не асинхронные. Нам нужно выключать защиту csrf что создает серьёзные уязвимости в сайте

`Web.php` предназначен для интерфейса с веба (не асинхронный интерфейс)

На самом деле роуты нужно прописывать в `api.php` и действовать нужно через него

И не нужно выключать csrf

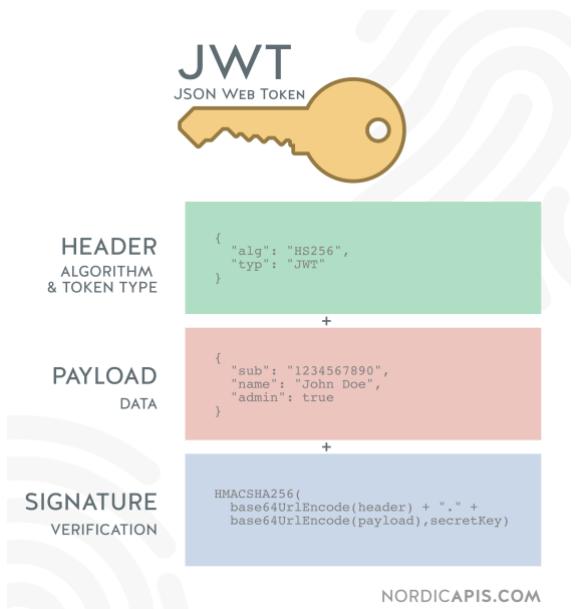


Тут

НО, ПРЕЖДЕ ЧЕМ ИСПОЛЬЗОВАТЬ ЭТИ РОУТЫ НУЖНО ПОДКЛЮЧИТЬ jwt token (джот токен)

Юзер запрашивает токен посредством отправки на роут логина логин и пароль – в ответ получает jwt токен – вставляет – можно использовать роуты api.php

JSON Web Token — это открытый стандарт для создания токенов доступа, основанный на формате JSON. Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях.



### Подключим токен

Нужно открыть папку vendor/tymon/Tymon\JWTAuth\Contracts\JWTSubject;

<https://jwt-auth.readthedocs.io/en/develop/laravel-installation/> - сайт

Тупо следуем указаниям которые там есть и все

```
composer require tymon/jwt-auth --ignore-platform-reqs (самая первая команда)
```

далее переходим сюда и делаем тут по списку

<https://jwt-auth.readthedocs.io/en/develop/quick-start/>

Тут мы первым делом подключаем интерфейсы, которые обязательны к применению

Для получения токена ПОД АККАУНТ НУЖНО АВТОРИЗИРОВАТЬСЯ

РЕГА = 1 АК = 1 ТОК

The screenshot shows a Postman interface with the following details:

- URL: `http://127.0.0.1:8000/api/auth/login?email=m@mail.ru&password=2021990qwe`
- Method: POST
- Headers: (8)
- Body: Pre-request Script, Tests, Settings, Cookies
- Params: email (checked), password (checked)
- Query Params: Key, Value, Description

Наш запрос (внимание на .../api/auth/...)

The screenshot shows the JSON response from the authentication request:

```
1 {
2     "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjE6ODAwMFwvYXBpXC9hdXRoXC9sb2dpbiIsIm1hdCI6MTYxODQ5NzIg3MmRiN2E1OTc2ZjcifQ.NqofLdWDbatD4pQgh8cAtaU0feQ7TL_k6Ru8lDNtxkQ",
3     "token_type": "bearer",
4     "expires_in": 3600
5 }
```

Получаем токен

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOl8vMTI3LjAuMC4xOjgwMDAvYXBpL2F1dGgvbG9naW4iLCJpYXQiOjE2NjQzMjEsImV4cCI6MTY2NDEyODMyMSwibmJmljoxNjY0MTI0NzIxLCJqdGkiOiwc3Fza1RINGRsT1hQWHNsliwic3ViljoiMilsInBydil6ljlzYmQ1Yzg5NDlmNjAwYWRI MzIINzAxYzQwMDg3MmRiN2E1OTc2ZjcifQ.xhwYR Oj2ZASm1Je5jX5opmDoAKjHb4li5fSF19zDe4o",
    "token_type": "bearer",
    "expires_in": 3600
}
```

ЧТО ДЕЛАТЬ С ТОКЕНОМ?

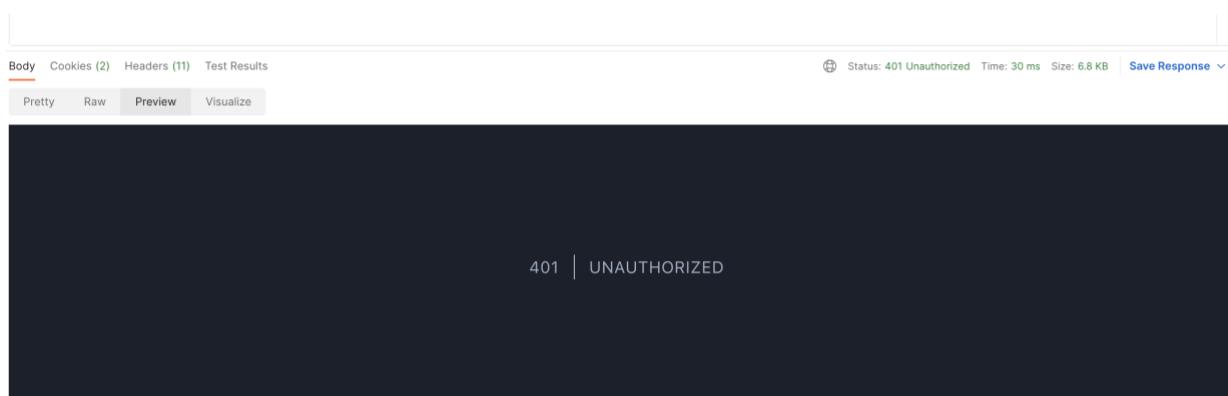
The screenshot shows the PhpStorm IDE interface. The top navigation bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help, and First\_project - api.php. The left sidebar displays the project structure under 'Project': first\_project (Resource root), app (Console, Exceptions, Http (Controllers, Post (AdminController, CreateController, DestroyController, EditController, IndexController, ShowController, UpdateController, AboutController, AutoController, ContactController, Controller, HomeController, MainController, MyPlaceController, PostController), Filters, Middleware, Requests). The main code editor window shows a portion of the RouteServiceProvider.php file:

```
23     'middleware' => 'api',
24     'prefix' => 'auth'
25 ],
26 ], function ($router) {
27
28     Route::post('login', 'AuthController@login');
29     Route::post('logout', 'AuthController@logout');
30     Route::post('refresh', 'AuthController@refresh');
31     Route::post('me', 'AuthController@me');
32
33 });
34 Route::group(['namespace' => 'Post', 'middleware' => 'jwt.auth'], function () {
35     Route::get('/posts', 'IndexController');
36 });
37
```

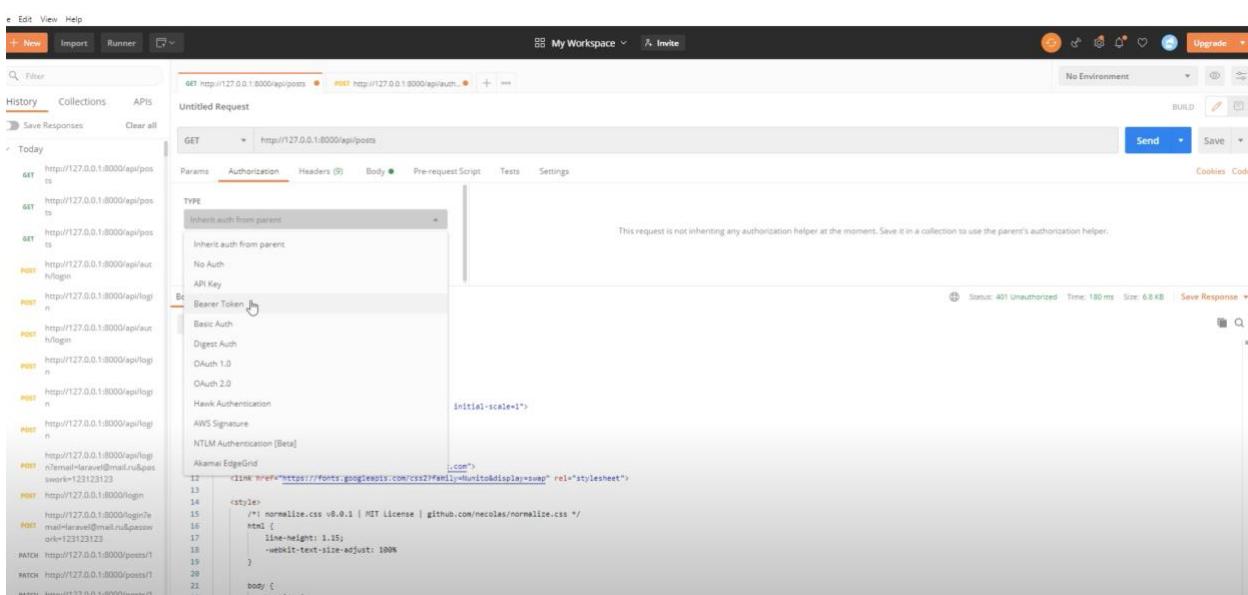
The bottom terminal window shows log output from a server process:

```
[Thu Apr 15 17:38:35 2021] 127.0.0.1:53160 Closed without sending a request; it was probably just an unused speculative preconnection
[Thu Apr 15 17:38:35 2021] 127.0.0.1:53160 Closing
[Thu Apr 15 17:38:36 2021] 127.0.0.1:53223 Accepted
[Thu Apr 15 17:38:36 2021] 127.0.0.1:53223 Closing
[Thu Apr 15 17:39:58 2021] 127.0.0.1:53297 Accepted
[Thu Apr 15 17:39:58 2021] 127.0.0.1:53297 Closing
[Thu Apr 15 17:40:11 2021] 127.0.0.1:53311 Accepted
[Thu Apr 15 17:40:11 2021] 127.0.0.1:53311 Closing
```

Прописываем роут теперь не в веб а в апи след образом



(если не подключить посредника jwt выдаст ошибку авторизации)



Далее делаем вот такой запрос и Авторизуемся Полученным токеном

И никакой csrf токен отключать не надо, все работает как надо

См видео 38

## CRUD с транзакцией

Транзакция (англ. transaction) — группа последовательных операций с базой данных, которая представляет собой логическую единицу работы с данными.

### Транзакции

Транзакция – набор операций (изменений), который должен быть выполнен полностью или не выполнен совсем (единий логический блок).

Транзакции параллельны, если их выполнение пересекается во времени.

Механизм транзакций – основа обеспечения целостности БД.

Свойства транзакции (**ACID**):

- Atomicity – атомарность,
- Consistency – согласованность,
- Isolation – изолированность,
- Durability – устойчивость.

### Понятие транзакции

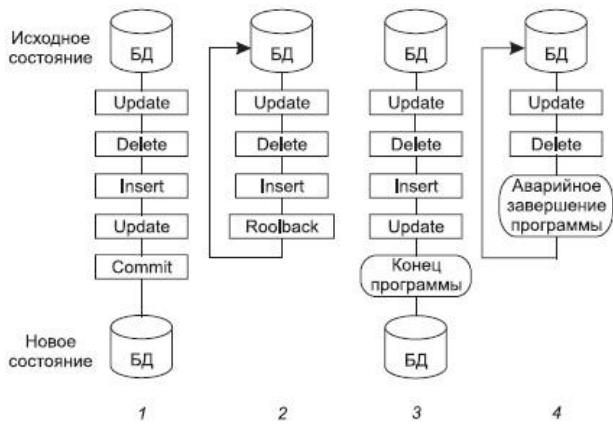
Транзакция – это последовательность операций, проводимых над БД, выполняемых как единое целое и переводящих БД из одного непротиворечивого состояния в другое непротиворечивое состояние

Количество операций, входящих в транзакцию, может быть любым от одной до сотен, тысяч

Разработчик решает, какие команды должны выполняться как одна транзакция, а какие могут быть разбиты на несколько последовательно выполняемых транзакций.

При выполнении транзакции СУБД должна обеспечить обработку набора команд, входящих в транзакцию, так, чтобы гарантировать правильность и надежность работы системы.

Транзакция должна удовлетворять ACID – требованиям



CRUD с транзакцией – создание тегов, постов, классов и т.д. в ОДНОМ ПОТОКЕ, это обеспечит защиту от ошибок и целостность БД. Избавление от разорванной связи

API.php – это роуты только для взаимодействия с вебом. Пользователи там не бывают

**ДАЛЬШЕ БУДЕТ ПОЗЖЕ**