Фреймворк Laravel

Что необходимо

- **IDE**
  - **PhpStorm**
  - **Atom**
  - **Sublime Text**
  - **Notepad++**
  - **И др.**
- https://getcomposer.org/ установка (обновление) Laravel, добавление (обновление) дополнительных пакетов в веб-проект.
- Установить laravel https://laravel.com/docs/5.4/installation
- **Homestead**

Требования

- PHP> = 5.6.4
- Расширение OpenSSL PHP
- Расширение PDO PHP
- Расширение Mbstring PHP
- Расширение Tokenizer PHP
- Расширение XML PHP

Установка

- composer global require "laravel/installer«
- Обязательно поместите каталог (или эквивалентный каталог для вашей ОС) в $ PATH, чтобы исполняемый файл мог быть обнаружен в вашей системе.$HOME/.composer/vendor/binlaravel

Конфигурация

- Публичный каталог
  - После установки Laravel вы должны настроить корневой документ / корневой public каталог веб-сервера в качестве каталога. В этом каталоге используется фронт-контроллер для всех HTTP-запросов, поступающих в ваше приложение.index.php
- Файлы конфигурации
  - Все файлы конфигурации для фреймворка Laravel хранятся в config каталоге. Каждый параметр задокументирован, поэтому не стесняйтесь просматривать файлы и знакомиться с доступными вам вариантами.
- Разрешения каталога
  - После установки Laravel вам может потребоваться настроить некоторые разрешения. Каталоги в пределах storage и каталоги должны быть доступны для записи вашего веб - сервера или Laravel не будет работать. Если вы используете виртуальную машину Homestead , эти разрешения уже должны быть установлены.bootstrap/cache

Конфигурация

- Ключ приложения
  - Следующее, что вам нужно сделать после установки Laravel, - это установить для ключа приложения случайную строку. Если вы установили Laravel через Composer или установщик Laravel, этот ключ уже был установлен для вас командой.php artisan key:generate
  - Обычно эта строка должна состоять из 32 символов. Ключ можно установить в .envфайле окружения. Если вы не переименовали файл в , сделайте это сейчас. Если ключ приложения не установлен, ваши пользовательские сеансы и другие зашифрованные данные не будут в безопасности!.env.example.env

Laravel Mix

- **CSS, JS, Less, Saas, Stylus, PostCSS**
- <link rel="stylesheet" href="{{ mix('/css/app.css') }}">

Laravel Forge

- **Laravel Forge – A**втоматическое развёртывания из Git
- **DigitalOcean**, **Linode**

СТРУКТУРА LARAVEL

- app — место, где хранятся наши контроллеры, модели
- bootstrap — системная папка Laravel
- config — настройки компонентов и модулей
- database — работа с базами (миграции, фабрики и дефолтные данные)
- public — общая точка входа (а также библиотеки JS, стили CSS, картинки)
- resources — хранит представления (VIEW), локализацию (перевод), а также библиотеки JS и стили CSS/SCSS
- routes — роутинг (маршрутизация)
- storage — здесь хранятся сессии, логи, кеш а также загруженные файлы
- tests — автотесты (phpUnit)
- vendor — загруженные библиотеки (composer)

СТРУКТУРА LARAVEL

- .env — файл с настройками “рабочей среды”
- .gitignore — настройки для работы с git репозиторием
- artisan — консольный помощник
- composer.json — настройки composer-зависимостей
- package.json — настройки npm-зависимостей для JS-библиотек
- phpunit.xml — уже подключена библиотека PHPUnit
- serve.php — скрипт для запуска проекта из корня

НАСТРОЙКА LARAVEL

2 варианта первоначальной настройки:

- Прописать все настройки внутри config/app.php, config/database.php
- Создать новый файл среды .env в котором прописать настройки

НАСТРОЙКА LARAVEL

В начале Laravel подтягивает настройки среды запуска (из .env)
и только потом ищет нужную директиву внутри папки config

пример из config/app.php:

```
'name' => env('APP_NAME',
'Laravel'),
```

ENVIRONMENT: РАБОЧАЯ СРЕДА LARAVEL

Назначение рабочей среды:

- разделять настройки для разработки/тестирования/production
- изолировать свои настройки рабочей среды от общего репозитория

Для создания своих настроек рабочей среды нужно заполнить файл **.env**.

Если его нет - скопировать **.env.example**

НАСТРОЙКА РАБОЧЕЙ СРЕДЫ LARAVEL 5

- APP_ENV — название рабочей среды
- APP_KEY — ключ приложения
- APP_DEBUG — дебаг включен/выключен
- APP_URL — полный url где размещено приложение. Laravel с помощью этой директивы строит пути
- DB_CONNECTION — драйвер соединения с БД (MySQL, MSSQL, Postgres, SQLite)
- DB_HOST — хост БД
- DB_PORT — порт соединения с БД
- DB_DATABASE — название схемы БД
- DB_USERNAME — пользователь БД
- DB_PASSWORD — пароль пользователя БД (если отсутствует - стираем директиву)

ARTISAN: ОСНОВНЫЕ КОМАНДЫ

Список команд:

```
php artisan list
```

Список команды определенного блока:

```
php artisan list make
```

Название блока указано со сдвигом влево, и это не является командой.

ARTISAN: ОСНОВНЫЕ КОМАНДЫ

Используемая рабочая среда:

```
php artisan env
```

Используемая версия Laravel:

```
php artisan --version
```

Генерация ключа приложения:

```
php artisan key:generate
```

ARTISAN: ОСНОВНЫЕ КОМАНДЫ

Выключение приложения:

```
php artisan down
```

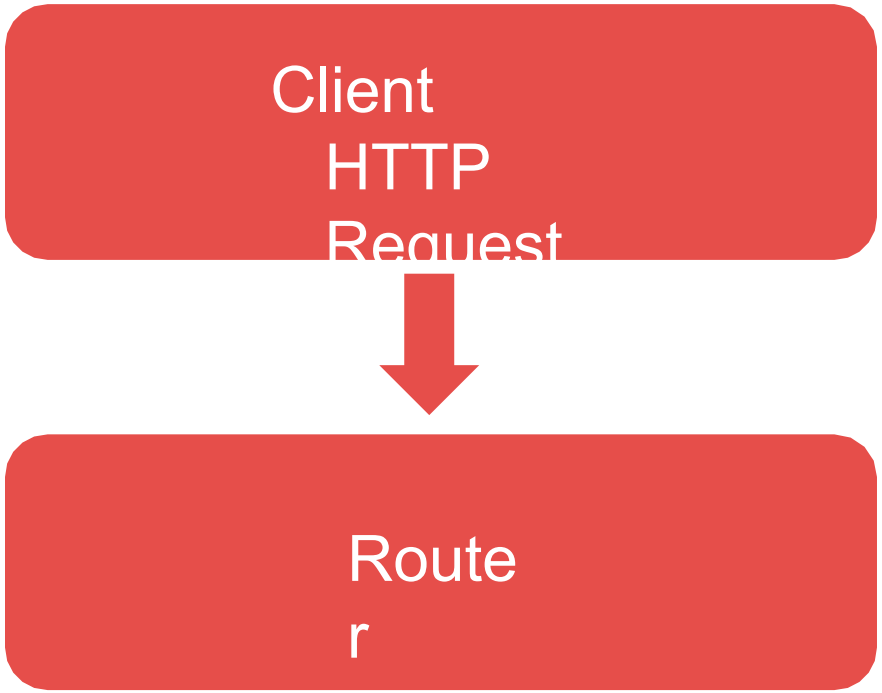
Включение приложения:

```
php artisan up
```

Запуск миграций:

```
php artisan migrate
```

МАРШРУТИЗАЦИЯ



МАРШРУТИЗАЦИЯ: РЕГИСТРАЦИЯ МАРШРУТОВ

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

- метод get говорит о том, что запрос пришел по методу GET на url “/”, то есть на главную страницу.
- вторым параметром указывается действие маршрута.

МАРШРУТИЗАЦИЯ: ROUTER

Поддерживаемые методы для регистрации маршрутов:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri,
$callback);
```

МАРШРУТИЗАЦИЯ: ROUTER

Иногда нужно перечислить сразу несколько методов для маршрута:

```
Route::match(['get', 'post'], '/', function () {
    //
});
```

Или использовать любой метод для маршрута:

```
Route::any('foo', function () {
    //
```

МАРШРУТИЗАЦИЯ: ПАРАМЕТРЫ МАРШРУТОВ

```
Route::get('user/{id}', function ($id) {
    return 'User '.$id;
});
```

В данном случае все, что будет стоять после user/ будет попадать в параметр id. Он же будет передаваться в значении переменной $id:

http://localhost/user/7 => User 7

МАРШРУТИЗАЦИЯ: ПАРАМЕТРЫ МАРШРУТОВ

- передаются маршруты в том же порядке, в котором они стоят внутри маршрута.
- название параметра внутри маршрута никак не связано с названием переданной переменной.

```
Route::get('posts/{post}/comments/{comment}', function ($postId, $commentId) {
    //
});
```

http://localhost/posts/6/comments/3 => [$postId => 6, $commentId => 3]

МАРШРУТИЗАЦИЯ: ПАРАМЕТРЫ МАРШРУТОВ

Параметры у маршрута могут быть необязательными: указываются с “?”.

```
Route::get('user/{id?}', function ($id = null)
    {  return 'User '.$id;
});
```

http://localhost/user => User

http://localhost/user/7 => User 7

МАРШРУТИЗАЦИЯ: ПРИВЯЗКА К ДЕЙСТВИЮ

2й параметр в регистрации маршрутов — привязка к действию. Действия (action) — это методы контроллера.

```
Route::get('about',

'PageController@about');  или

Route::get('about', ['uses' =>
```

МАРШРУТИЗАЦИЯ: ПРИВЯЗКА К ДЕЙСТВИЮ

Для одного маршрута должно быть зарегистрировано одно действие.

МАРШРУТИЗАЦИЯ: ПРИОРИТЕТ МАРШРУТОВ

Каждый следующий маршрут перекрывает предыдущий, если они пересекаются по url:

```
Route::get('route', 'FirstController@action');

Route::get('route', 'SecondController@action');
```

http://localhost/route => SecondController@action

МАРШРУТИЗАЦИЯ: ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Это позволит упростить роутер и повысить читаемость кода.

Появились методы view() и redirect() с версии Laravel 5.5

```
Route::view('/', 'welcome');

Route::redirect('/here', '/there', 301);
```

МАРШРУТИЗАЦИЯ: ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Указание параметра маршрута через регулярные выражения:

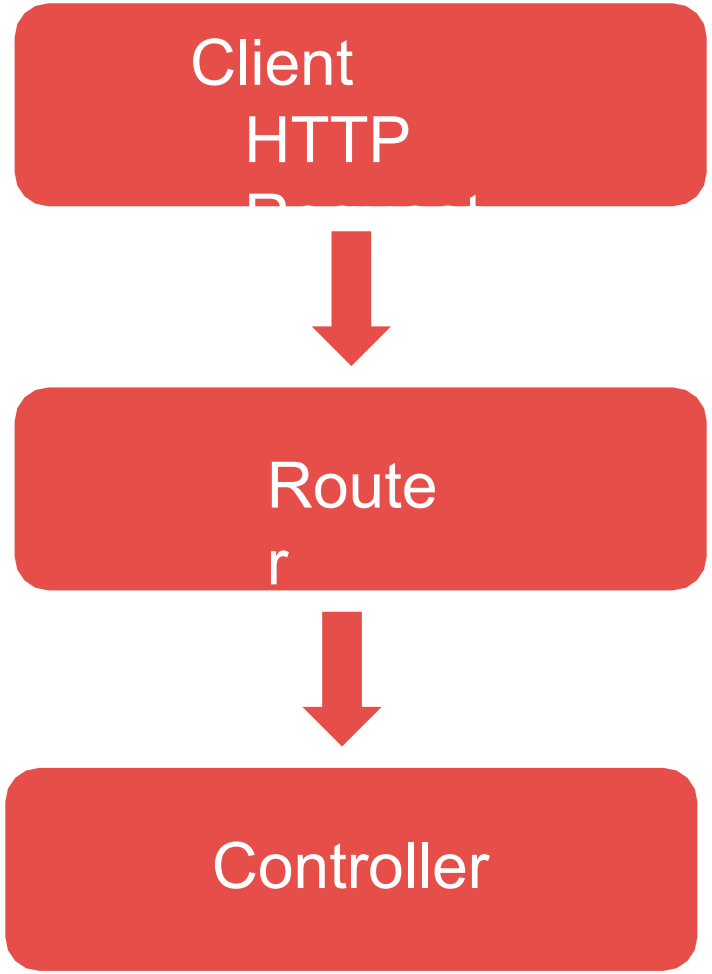
```
Route::get('user/{name}', function ($name) {
        //
})->where('name', '[A-Za-z]+');

Route::get('user/{id}', function ($id) {
        //
})->where('id', '[0-9]+');
```

КОНТРОЛЛЕРЫ

Контроллеры — это классы, которые содержат (организовывают) логику обработки запроса, работу с моделью данных и представлением.

Контроллеры обычно хранятся в папке app/Http/Controllers

КОНТРОЛЛЕРЫ



КОНТРОЛЛЕРЫ: СОЗДАНИЕ

Создание контроллера:

```
php artisan make:controller UserController
```

В конце метода обычно возвращается представление через view:

```
return view('welcome');
```

КОНТРОЛЛЕРЫ: СОЗДАНИЕ

Возможно создание контроллера в нужной области видимости внутри директории app/Http/Controllers:

```
php artisan make:controller Admin/PageController
```

КОНТРОЛЛЕРЫ: ПАРАМЕТРЫ МАРШРУТА

- каждый переданный в маршруте параметр требуется обработать в контроллере
- значения параметров передаются в контроллер в том же порядке, в каком они заданы в маршруте
- необязательному параметру в методе ему нужно указать значение по-умолчанию

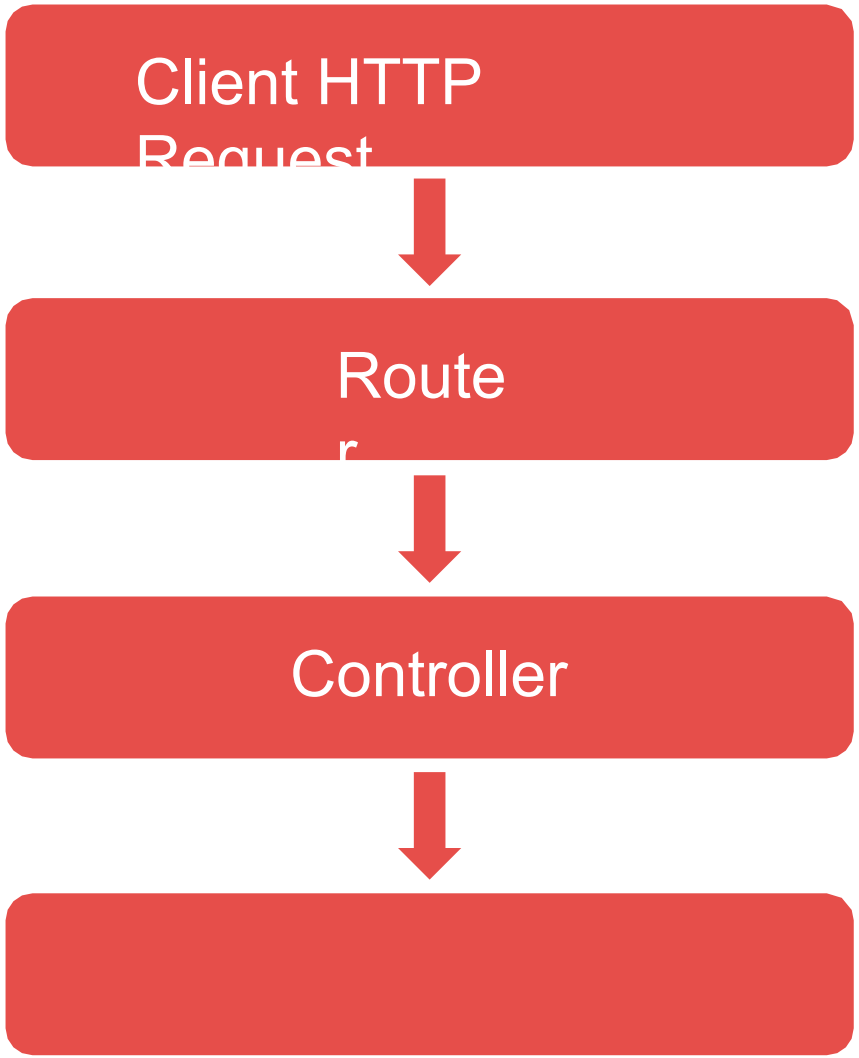
КОНТРОЛЛЕРЫ: ПАРАМЕТРЫ МАРШРУТА

```php
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class PagesController extends Controller
{
    public function user($userId)
    {
        //$userId - это обязательный параметр
    }
    public function news($page = null)
    {
      //$page - необязательный параметр
    }
```

BLADE: ПРЕДСТАВЛЕНИЯ

- Для удобства работы со страницами в Laravel существуют представления (view). Они находятся в папке **resources/views**.
- В Laravel по-умолчанию используется свой шаблонизатор **Blade**, но можно подключить twig или другой шаблонизатор.
- В представлениях обычно используется HTML-код или Markdown-разметка, управляющие конструкции и переданные данные.

ПРЕДСТАВЛЕНИЯ



BLADE: ПОДКЛЮЧЕНИЕ ПРЕДСТАВЛЕНИЙ

- Представления подключаются через **view()**, где первый параметр - название представления, а второй - переданные в него данные.
- Имя представления нужно указывать **без .blade.php**
- Для удобства можно разнести шаблоны по различным директориям. Чтобы Blade их нашел, нужно в названии представления указать весь путь, **используя точку или слеш**.

BLADE: ПЕРЕДАЧА ДАННЫХ В ПРЕДСТАВЛЕНИЕ

```
view('welcome', ['data' => $data]);
view('welcome', compact('data'));
view('welcome')->with(['data' =>
$data]);
```

Пример подразумевает наличие переменной $data в методе контроллера, и переменная будет доступна в представлении.

BLADE: ОТОБРАЖЕНИЕ ДАННЫХ В ШАБЛОНЕ

- Для отображения значения переменной в представлении используется **{{ $var }}**
- Переменная обрабатывается функцией htmlentities()
- Если нужно отобразить значение переменной без этой функции, нужно использовать **{!! $var !!}**
- Если конструкция {{ }} на странице нужна для работы с js-фреймворками, можно использовать **@{{ $var }}**, чтобы Blade не обрабатывал скобки, а выводил как есть.

BLADE: УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

Blade поддерживает следующие управляющие конструкции:

if, if … else, isset, empty, for, foreach, while

- указываются через @
- используются с закрывающим тегом (end + название директивы)

```
@foreach($users as $user)
    {{ $user->name
}}  @endforeach
```

BLADE: УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

```
@isset($user)

    @foreach($users as $user)
        <p>{{ $user->name }}
        @if($user->approved()
        )
            Пользователь подтвержден <br>
        @endif
        </p>
    @endforeac
    h
```

ДЕБАГ ПАНЕЛЬ
BARRYVDH

Полезный инструмент для разработчика: пакет **barryvdh/laravel-debugbar**

```
composer require barryvdh/laravel-debugbar
```

В конфиг config/app.php прописать в секцию providers:

Barryvdh\Debugbar\ServiceProvider::class,

ДЕБАГ ПАНЕЛЬ BARRYVDH

Для среды production убрать вывод ошибок и информацию для дебага:

В **app\Providers\AppServiceProvider.php** в методе **register()**:

```
if (env('APP_DEBUG')) {
        $this->app->register('Barryvdh\Debugbar\ServiceProvider');
}
```

Теперь, если у приложения в среде установлена директива **APP_DEBUG** в значение **false** — работать дебаг панель не будет.

РАБОТА СО СТРУКТУРОЙ БАЗЫ ДАННЫХ

Решение №1

dump

файл

Проблемы:

- при каждом обновлении этого файла придётся запускать этот файл целиком
- невозможность сохранить данные после обновления
- недостаточная автоматизация процесса

РАБОТА СО СТРУКТУРОЙ БАЗЫ ДАННЫХ

Решение №2

Разбить dump-файл на отдельные файлы, с каждым изменением добавлять новые файлы, что позволит сохранить данные.

Проблемы:

- нет информации о последовательности запуска файлов
- нет информации о том, какой был запущен, а какой - нет
- недостаточная автоматизация процесса

РАБОТА СО СТРУКТУРОЙ БАЗЫ ДАННЫХ

Решение №3

Добавить timestamp в названия файлов и создать таблицу, где записывать выполненные инструкции

Проблемы:

- невозможность вернуться в нужное состояние
- недостаточная автоматизация процесса

РАБОТА СО СТРУКТУРОЙ БАЗЫ ДАННЫХ

Решение №4

Для каждого файла создавать файл-дублёр, который позволит вернуться на шаг назад

Проблемы:

- недостаточная автоматизация процесса
- сложность решения

МИГРАЦИИ: ЧТО ТАКОЕ МИГРАЦИИ

Миграции — встроенный механизм в Laravel, позволяющий решить все вышеперечисленные проблемы.

Миграция представляет из себя класс, имеющий 2 метода: **up()** и **down()**

МИГРАЦИИ: ЧТО ТАКОЕ МИГРАЦИИ

Метод **up()** содержит в себе инструкции, которые запускаются при запуске миграции.

Метод **down()** содержит в себе инструкции, которые позволяют вернуть систему в исходное состояние.

Эти 2 метода «зеркальные» относительно друг друга.

МИГРАЦИИ: СОЗДАНИЕ

Создаются миграции через artisan:

```
php artisan make:migration
```

create_new_table — название миграции.

- создаст файл timestamp_create_new_table.php
- создаст класс CreateNewTable

имена миграций не должны повторяться.

МИГРАЦИИ: СОЗДАНИЕ

дополнительные флаги:

--create=tableName — создание таблицы tableName

--table=tableName — изменение таблицы tableName

МИГРАЦИИ: СОЗДАНИЕ

- Schema::create — создание таблицы
- Schema::table — обновление таблицы
- Schema::drop — удаление таблицы

create & table имеют 2 параметра:

1. название таблицы
2. функция-замыкания с действиями над таблицей (передается Blueprint, используется в переменной $table)

МИГРАЦИИ: СОЗДАНИЕ

increments('id') — создает автоинкрементное поле с названием id

timestamps() — создает 2 поля created_at и updated_at, которые уже задействованы в Laravel

МИГРАЦИИ: СОЗДАНИЕ

Класс Blueprint содержит:

- методы, названные по типу поля, которое оно создает (**bigInteger**, **char**, **date**, **integer**, **longText**, **string**)
- дополнительные методы, которые кроме типа поля определяют дополнительные свойства и/или имя поля (**increments**, **bigIncrements**, **ipAddress**, **rememberToken**)
- методы, которые могут создавать сразу несколько полей (**timestamps**)

МИГРАЦИИ: СОЗДАНИЕ

В методах создания полей параметры:

1. название поля
2. длина поля, массив значения (для **enum**)
3. для полей типа **decimal**, **double**, **float**, указывается число знаков после запятой

Дополнительными методами указываются:

- дополнительные свойства (**nullable, unsigned**)
- положение относительно другого столбца (**after, first**)
- комментарий (**comment**)
- значение по-умолчанию (**default**).

МИГРАЦИИ: ЗАПУСК

```
php artisan migrate
```

- миграции будут запущены в порядке создания
- если это первый запуск миграции, то в базе данных будет создана таблица **migrations**, куда запишутся наши миграции

МИГРАЦИИ: ЗАПУСК

Выполнятся все не выполненные ранее миграции.

- После того, как миграции будут выполнены — в таблице migrations появятся записи.
- Кроме названия файла миграции есть ещё поле **batch**, в которое последовательно записывается число — порядковый номер пакета миграций.

МИГРАЦИИ: ЗАПУСК

- *SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 767 bytes*

- **app/Providers/AppServiceProvider.php** в методе

boot прописать: Schema::defaultStringLength(191);

- При этом не забыть указать

использование **Schema**: use

Illuminate\Support\Facades\Schema;

МИГРАЦИИ: ОТКАТ

откат миграций:

```
php artisan migrate:rollback
```

откат миграций только конкретного количества миграций:

```
php artisan migrate:rollback --step=2
```

МИГРАЦИИ: ОТКАТ

откатить сразу все миграции

```
php artisan migrate:reset
```

выполнить откат и запуск всех миграций сразу:

```
php artisan migrate:refresh
```

МИГРАЦИИ: ВНЕСЕНИЕ ИЗМЕНЕНИЙ

При необходимости внести изменения в миграцию (таблицу), если она уже отправлена в git, нужно создать новую миграцию.

МИГРАЦИИ: МИНУСЫ МЕХАНИЗМА

При переключении между ветками в git может возникнуть проблема:

в базе данных миграция указана как выполненная, а миграции нет в данной ветке.

ЗАПРОСЫ В БАЗУ ДАННЫХ

Написание запросов:

```
$users = DB::select('select * from users');
$results = DB::select('select * from users where id = :id', ['id' =>
1]);
```

Результат возвращается массивом.

Выполнение запроса, не возвращающего результата:

```
DB::statement('drop table users');
```

КОНСТРУКТОР ЗАПРОСОВ В БД

```
$user = DB::table('users')->where('name', 'John')->first();
```

where() имеет 2 или 3 параметра.

- Если параметра только 2, то вызов эквивалентен where(name', '=', 'John').
- Если параметра 3, то 2м параметром указывается знак сравнения.

Метод **first**() возвращает первый элемент из возвращаемой **коллекции**.

КОНСТРУКТОР ЗАПРОСОВ В БД

Выбрать все значения из таблицы можно методами **all()** и **get()**:

```
$users = DB::table('users')->all();
$users = DB::table('users')->get();
```

КОНСТРУКТОР ЗАПРОСОВ В БД

- value() возвращает одно значение:
- $email = DB::table('users')->where('name', 'John')->value('email'); pluck()

возвращает коллекцию - целиком или один столбец:

- $titles = DB::table('roles')->pluck('title'); или пару ключ => значение:

- $roles = DB::table('roles')->pluck('title', 'name');

КОНСТРУКТОР ЗАПРОСОВ В БД

Кроме того, возможно задавать, какие конкретно поля нужно выбирать:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

В конструкторе присутствует метод distinct():

```
$users = DB::table('users')->distinct()->get();
```

Агрегирующие запросы:

```
$users = DB::table('users')->count();
$price = DB::table('orders')->max('price');
```

КОНСТРУКТОР ЗАПРОСОВ В БД

В конструкторе также реализована поддержка как innerJoin:

```
$users = DB::table('users')
->join('contacts', 'users.id', '=', 'contacts.user_id')
->select('users.*', 'contacts.phone')
->get();
```

так leftJoin:

```
$users = DB::table('users')
->leftJoin('posts', 'users.id', '=', 'posts.user_id')
->get();
```

КОНСТРУКТОР ЗАПРОСОВ В БД

А также «перемножения» таблиц:

```
$users =
DB::table('sizes')->crossJoin('colours')->get();
```

Доступны и методы **groupBy()** и **having()**:

```
$users = DB::table('users')
->groupBy('account_id')
->having('account_id', '>', 100)
->get();
```

КОНСТРУКТОР ЗАПРОСОВ В БД

- orWhere('fieldName', 'value1')
- whereBetween('fieldName', ['value1', 'value2'])
- whereNotBetween('fieldName', ['value1', 'value2'])
- whereIn('fieldName', ['value1', 'value2', ...])
- whereNotIn('fieldName', ['value1', 'value2', ...])
- whereNull('fieldName')
- whereNotNull('fieldName')

КОНСТРУКТОР ЗАПРОСОВ В БД

Сортировка методом **orderBy()**:

```
$users = DB::table('users')->orderBy('sort');
$users = DB::table('users')->orderBy('sort',
desc,);
```

КОНСТРУКТОР ЗАПРОСОВ В БД

Конструктор поддерживает построение запросов и для INSERT, UPDATE, DELETE.

```
DB::table('users')->insert(['email' => 'john@example.com', 'votes' =>
0]);  DB::table('users')->where('id', 1)->update(['votes' => 1]);
DB::table('users')->delete();
```

КОНСТРУКТОР ЗАПРОСОВ В БД

Для того, чтобы посмотреть, какой запрос формирует конструктор запросов, есть метод **toSql()**:

```
$posts = DB::table('posts')->where('id', '>',
15)->orderBy('sort');  echo $posts->toSql();
```

*SELECT * FROM `posts` where `id` > ? order by `sort` asc*

КОЛЛЕКЦИИ

Результат выполнения запроса — коллекция. Коллекция — это объект, содержащий набор элементов. Для создания коллекции: **collect()** или **new Collection();**

КОЛЛЕКЦИИ

Все данные в коллекции лежат в свойствах объектов.

```
$user = DB::table('users')->where('name', 'John')->first();
```

Свойство **name** будет доступно через свойство **name** экземпляра объекта **Collection** в переменной **$user**:

```
echo $user->name;
```

КОЛЛЕКЦИИ: НАИБОЛЕЕ ПОЛЕЗНЫЕ МЕТОДЫ

- count()
- toArray()
- toJson()
- first()
- last()
- filter()
- map() — проходит по всем элементам коллекции, выполняя действия с каждым
- sort() — отсортировывает элементы коллекции по значению
- sortBy() — отсортировывает элементы коллекции по полю

БАЗА ДАННЫХ:
SEEDER

Первичные данные задаются через Seeder.

Seeder — это класс с единственным методом **run()**, который выполняется при запуске.

- наследуются от **Illuminate\Database\Seeder**
- лежат в папке **database\seeds**

```
php artisan make:seeder
```

В методе run() содержатся запросы для базы данных.

SEEDER: ЗАПУСК

```
php artisan
```

```
php artisan migrate
```

```
php artisan db:seed
```

При этом, каждый seeder должен быть подключен в файле **database\seeds\DatabaseSeeder.php** в методе **run()**:

```
$this->call(MySeeder::class);
```

SEEDER: ЗАПУСК

Недостаток: нет информации, запускался ли Seeder.
Решение: Подключить seeder в миграцию

```
(new \MySeeder())->run();
```

МОДЕЛИ

- Модель – это класс, используемый для описания некоторой сущности.
- Модель позволяет абстрагироваться от способа хранения и обработки данных, и заниматься непосредственно бизнес-логикой приложения.

МОДЕЛИ
ELOQUENT

Eloquent — красивая и простая реализация шаблона ActiveRecord в Laravel для работы с базами данных.

- Каждая таблица имеет свой класс-модель, который используется для работы с этой таблицей;
- Каждый экземпляр данного класса соответствует одной записи таблицы;
- Модели позволяют читать/писать/обновлять/удалять данные в таблицах;
- Все модели Eloquent наследуют класс **Illuminate\Database\Eloquent\Model**.

МОДЕЛИ ELOQUENT: СОЗДАНИЕ

Создание модели:

```
php artisan make:model Name
```

Name — название модели в единственном числе.
Модель связана с таблицей **names** (множественное число).

Создание модели с миграцией:

```
php artisan make:model Name
```

МОДЕЛИ ELOQUENT: СОЗДАНИЕ

Создание модели в другой области видимости:

```
php artisan make:model Admin/User
```

Свойство **$table** переопределяет связанную таблицу:

Свойство **$primaryKey** переопределяет первичный ключ, если требуется использовать не id.

МОДЕЛИ ELOQUENT: СОЗДАНИЕ

```
<?php
namespace
App;
use Illuminate\Database\Eloquent\Model;
class Name extends Model
{
    protected $table = 'names_table';
    protected $primaryKey =
    'name_id';
}
```

МОДЕЛИ ELOQUENT: TIMESTAMPS

При работе с таблицей через модель, при создании записи в поле created_at и updated_at записывается значение текущего времени.
При обновлении записи — обновляется поле updated_at.

Если нужно, чтобы Eloquent не работал с этими полями:

protected $timestamps = false;

МОДЕЛИ ELOQUENT: TIMESTAMPS

Формат значения полей created_at & updated_at задается свойством **$dateformat**

Константы CREATED_AT и UPDATED_AT определяют названия полей.

МОДЕЛИ ELOQUENT: ПОЛУЧЕНИЕ ЗАПИСЕЙ

Для получения всех записей из таблицы, используется метод all():

```
$posts = App\Post::all();
```

В переменной $posts коллекция экземпляров модели App\Post.

МОДЕЛИ ELOQUENT: КОНСТРУКТОР ЗАПРОСОВ

Eloquent позволяет строить запросы через конструктор запросов:

```
$posts = App\Post::where('status', '>=', 0)->get();
```

*SELECT * FROM posts WHERE status >= 0;*

Все что «слева» **get()** (или **all()** или **first()** и т.д.) — конструктор запросов. Методы, указанные после — методы коллекции.

```
$posts = App\Post::all()->where('status', '>=', 0);
```

*SELECT * FROM posts;*

МОДЕЛИ ELOQUENT: АГРЕГИРУЮЩИЕ ФУНКЦИИ

```
$count = App\Post::where('active',
1)->count();

$min = App\Post::min('updated_at');

$min = App\Post::max('id');
```

МОДЕЛИ ELOQUENT: ПОЛУЧЕНИЕ ЗАПИСЕЙ

Получить вместо коллекции одну первую запись:

```
$post = App\Post::first();
```

Получение записи по первичному ключу:

```
$post = App\Post::find($id);
```

```
App\Post::where('id', $id)->first();
```

МОДЕЛИ ELOQUENT: ПОЛУЧЕНИЕ ЗАПИСЕЙ

Получение нескольких записей по id (вернет коллекцию):

```
$post = App\Post::find([1, 2, 3]);
```

Если по одному из переданных значений ничего не будет найдено — ошибки вызвано не будет.

МОДЕЛИ ELOQUENT: ПОЛУЧЕНИЕ ЗАПИСЕЙ

Вызов 404 ошибки (Not Found Exceptions) при отсутствии записи:

```
$post = App\Post::findOrFail($id);
```

```
$post = App\Post::firstOrFail();
```

Оба метода возвращают один экземпляр модели:

в первом случае, если запись с id = $id

существует,

во втором — если существует хотя бы одна запись в таблице, при отсутствии других условий.

МОДЕЛИ ELOQUENT: ПОРЯДОК ЗАПИСЕЙ

Метод **orderBy()** работает с конструктором запросов.

```
$posts = Post::orderBy('sort')->get();
```

*SELECT * FROM posts ORDER BY sort*

Метод **sortBy()** работает с коллекцией.

```
$posts = Post::get()->sortBy('sort');
```

*SELECT * FROM posts*

МОДЕЛИ ELOQUENT: ПОРЯДОК ЗАПИСЕЙ

Сортировка в обратном порядке в конструкторе запросов:

```
$posts = Post::orderBy('sort', 'desc')->get();
```

```
$posts = Post::orderByDesc('sort')->get();
```

Сортировка коллекции в обратном порядке:

```
$posts = Post::all()->sortBy('id', 'desc');
```

МОДЕЛИ ELOQUENT: ДОСТУП К ЗНАЧЕНИЯМ

```
$posts = Post::all();


foreach ($posts as $post) {
        echo $post->title;
}
```

МОДЕЛИ ELOQUENT: СОЗДАНИЕ ЗАПИСИ

Создание новой записи — метод **save()**:

```
$post = new App\Post;
$post->title = 'Some title';
$post->body = 'Body text';
$post->save();
```

МОДЕЛИ ELOQUENT: СОЗДАНИЕ ЗАПИСИ

При использовании метода **create()**:

```
$post = App\Post::create([title' => 'Some title', body' => 'Body text']);
```

При этом в модели должны быть указаны заполняемые поля в свойстве **$fillable**.

Иначе будет вызвана ошибка **Mass Assignment Exception.**

При защите свойств от «массовой записи» используется свойство **$guarded**.

МОДЕЛИ ELOQUENT: ОБНОВЛЕНИЕ ЗАПИСИ

Для обновления записи нужно изменить свойства записи и вызвать метод **save()**:

```
$post = App\Post::find($id);
$post->title = New title';
$post->save();
```

МОДЕЛИ ELOQUENT: ОБНОВЛЕНИЕ ЗАПИСИ

Использование метода **update()**:

```
$post = App\Post::find($id)->update([title' => 'Some title']);
```

При этом поля, которые будут обновлены, должны быть указаны в свойстве **$fillable** или отсутствовать в свойстве **$guarded**.

МОДЕЛИ ELOQUENT: УДАЛЕНИЕ ЗАПИСИ

Удаление записи модели Eloquent методом **delete()**:

```
App\Post::find($id)->delete();
```

Метод **destroy()** позволяет удалить запись по $id:

```
App\Post::destroy($id);
```

```
App\Post::destroy(1, 2, 3);
```

```
App\Post::destroy([1, 2, 3]);
```

МОДЕЛИ ELOQUENT: SCOPE

Scope в Laravel («область запроса») — это некоторые условия выбора.

- помогают структурировать код и делают его понятнее для программиста.
- обозначаются методы в модели с приставкой scope и CamelCase.
- вызываются без приставки scope и с маленькой буквы.

МОДЕЛИ ELOQUENT: LOCAL SCOPE

Local scope уже содержат в себе конкретные условия выборки:

```
class User extends Model
{
    public function scopeActive($query)
    {
        return $query->where('is_active', 1);
    }
}
```

МОДЕЛИ ELOQUENT: LOCAL SCOPE

Воспользоваться данным методом можно через:

```
$users = User::active()->where('age', '>=', '18')->get();
```

В данном случае метод active() (обратите внимание, метод называется не *scopeActive()*) вернет **Query Builder**, который позволит дописать условия запроса.

МОДЕЛИ ELOQUENT: DYNAMIC SCOPE

Если в scope нужно передать какой-то динамический параметр — такие scope будут динамическими.

```
class User extends Model
{
    public function scopeType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

МОДЕЛИ ELOQUENT: АКСЕССОРЫ (GETTER)

Аксессоры — возможность изменить значение поля при обращении к нему вне модели:

```
function getFirstNameAttribute()
{
    return ucfirst($this->first_name);
}
```

```
$name = $user->first_name; //будет вызван getFirstNameAttribute()
```

МОДЕЛИ ELOQUENT: МУТАТОРЫ (SETTER)

Мутатор — метод, который вызывается перед заданием конкретного свойства модели:

```
function setFirstNameAttribute($value)
{
    $this->attributes['first_name'] = ucfirst($value);
}

$user->first_name; //получим результат setFirstNameAttribute()
```

CARBON

Для работы с датой/временем в Laravel подключена библиотека **Carbon**.

По умолчанию поля created_at и updated_at в модели Eloquent возвращают экземпляр класса Carbon.

При использовании класса Carbon в своих классах нужно объявлять его

use Carbon\Carbon;

CARBON

Carbon позволяет создавать дату (экземпляр класса Carbon) из входных параметров:

```
$dtToronto = Carbon::createFromDate(2012, 1, 1, 'America/Toronto');
$dtVancouver = Carbon::createFromDate(2012, 1, 1, 'America/Vancouver');
```

Вычислять разницу между различными датами:

```
$dtVancouver->diffInHours($dtToronto); // 3
```

и другие

CARBON

Для данной библиотеки важно установить верное время + часовой пояс в файле config/app.php в директиве **timezone**.

Полный список методов API можно найти на странице документации:

http://carbon.nesbot.com/docs/

BLADE: ПОДШАБЛОНЫ

@include

```
@include('field', ['name' => 'Age', value=> $age])
```

- первый параметр - blade-шаблон, который будет подключен
- второй параметр - переменные, которые будут переданы в подшаблон

BLADE: ПОДШАБЛОНЫ

@component

```
@component('components.footer', ['name' => 'Age', value=> $age])

...

@endcomponent
```

- 1й и 2й аргументы - идентичны как для @include
- Всё что между @component... @endcomponent - окажется в переменной $slot

BLADE: ПОДКЛЮЧЕНИЕ МАСТЕР ШАБЛОНА

Дочерний шаблон:

```
@extends('layout.master')
@section('title', 'Небольшую строку можно указать так')


@section('content')


Это контент внутренней страницы

@endsection
```

BLADE: ПОДКЛЮЧЕНИЕ МАСТЕР ШАБЛОНА

Мастер шаблон:

```
<html>
    <head>
        <title>@yield</title>
    </head>
    <body>
        @yield('content')
    </body>
</html>
```

- Для extends также можно передать 2м параметром переменные

ОБРАБОТКА ЗАПРОСА

Illuminate\Http\Request

- При работе с запросом подключаем класс 1м параметром.
- Остальные параметры указываются после него.
- Если не используется — можно убрать из параметров.

```
public function update(Request $request, $id)
{
    //...
}
```

ОБРАБОТКА ЗАПРОСА: ПОЛУЧЕНИЕ URL

Получить часть запроса без параметров uri:

```
$uri = $request->path();
```

Получить полную строку запроса с параметрами:

```
$url = $request->fullUrl();
```

Проверка url на соответствие:

```
$request->is('admin/*');
```

ОБРАБОТКА ЗАПРОСА: ПОЛУЧЕНИЕ МЕТОДА

Получение названия метода запроса:

```
$method = $request->method();
```

Проверка на соответствие используемого метода запроса:

```
if ($request->isMethod('post'))

{


}
```

ОБРАБОТКА ЗАПРОСА: ПОЛУЧЕНИЕ ДАННЫХ

Для получения переменной из запроса есть метод **input()**:

```
$name = $request->input('name');
```

Второй параметр позволяет задать значение по-умолчанию:

```
$name = $request->input('name',

'Franky');
```

Метод get() - синоним методу input():

```
$name = $request->get('name', 'Franky');
```

ОБРАБОТКА ЗАПРОСА: ПОЛУЧЕНИЕ ДАННЫХ

Получение данных запроса, через свойства объекта:

```
$name = $request->name;
```

Использование функции-хелпера **request()** вместо объекта класса Request:

```
$name = request()->name;

$title = request()->input('title');
```

ОБРАБОТКА ЗАПРОСА: ПОЛУЧЕНИЕ ДАННЫХ

Получение всех данные из запроса с помощью метода **all()**:

```
$input = $request->all();
```

Получение определённых параметров запроса через **only()**:

```
$input = $request->only('username',

'password');  Получить все параметры, кроме
```

- **except()**:

```
$input = $request->except('credit_card');
```

ОБРАБОТКА ЗАПРОСА: ПРОВЕРКА ДАННЫХ

Проверка существования переменной в запросе:

```
if ($request->has('name')) {


}
```

Возможно проверять сразу несколько переменных:

```
if ($request->has(['name', 'email'])) {
}
```

ОБРАБОТКА ЗАПРОСА: СОХРАНЕНИЕ ФАЙЛА

Проверка отправки файла в запросе:

```
request()->hasFile('image')
```

Получить путь к файлу можно через метод file():

```
$file = $request->file('image');
```

ОБРАБОТКА ЗАПРОСА: РАБОТА С ФАЙЛОМ

path() — получает путь сохраненного файла:

```
$request->file('image')->path();
```

extension() — получение расширения загруженного файла

```
$request->file('image')->extension();
```

getClientOriginalName() — получение исходного названия файла

```
$request->file('image')->getClientOriginalName();
```

ОБРАБОТКА ЗАПРОСА: СОХРАНЕНИЕ ЗАГРУЖЕННОГО ФАЙЛА

Сохранение загруженного файла в директории.

$request->file('image')->store($destination, $disk);

$destination — директория, в которую нужно сохранять файл.

$disk — отвечает за то, какой диск из настроек **Storage** выбран. Если параметр не указан — используется диск по-умолчанию.

STORAGE: РАБОТА С ФАЙЛАМИ

Позволяет использовать несколько хранилищ (дисков/подключений):

- локальное хранилище
- FTP
- сервис S3 (Amazon)
- сервис Rackspace

При этом с локальным хранилищем можно работать как в доступном по web (public) так и в недоступном месте.

STORAGE: НАСТРОЙКА ПОДКЛЮЧЕНИЙ

config/filesystems.php

default — подключение по-умолчанию

disks (массив) — сами подключения.

По умолчанию уже настроены local и public.

Файлы хранятся в папке **storage/app**. Папка **app** по умолчанию фигурирует в этих 2х подключениях.

STORAGE: НАСТРОЙКА ПОДКЛЮЧЕНИЙ

- driver (для работы с файловой системой используется драйвер local)
- root — основная папка подключения
- url — шаблон формирования публичных путей к файлам
- visibility — видны ли файлы подключения из вне через public

STORAGE: ОСНОВНЫЕ МЕТОДЫ

Чтобы воспользоваться методами Storage обязательно нужно указать его использование:

```
use Illuminate\Support\Facades\Storage;
```

По-умолчанию выбрано подключение default. Чтобы выбрать другое соединение, следует использовать метод disk():

```
Storage::disk('avatars')->put('file.jpg', $contents);
```

Метод put в данном случае сохраняет файл с содержимым $contents.

STORAGE: ОСНОВНЫЕ МЕТОДЫ

Получение файла:
$contents =

Storage::get('file.jpg');

Формирование пути (url):

$url = Storage::url('file1.jpg');

Получение размера

файла:

STORAGE: ОСНОВНЫЕ МЕТОДЫ

Получение даты последнего изменения файла:
$time = Storage::lastModified('file1.jpg');

Записать информацию в начало файла:

Storage::prepend('file.log', 'Prepended

Text'); Записать информацию в конец

файла: Storage::append('file.log',

'Appended Text');

STORAGE: ОСНОВНЫЕ МЕТОДЫ

Копирование файла:
Storage::copy('old/file1.jpg', 'new/file1.jpg');

Перемещение файла:

Storage::move('old/file1.jpg', 'new/file1.jpg');

Удаление файла:

Storage::delete('file.jpg');

STORAGE: ОСНОВНЫЕ МЕТОДЫ

При использовании методов copy(), move() - стоит помнить, что эти методы работают только внутри одного диска.

STORAGE: ОСНОВНЫЕ МЕТОДЫ

Создание и удаление директорий:
Storage::makeDirectory($directory);

Storage::deleteDirectory($directory)

;

STORAGE: ГЕНЕРАЦИЯ URL

Формирование пути к файлу для веб-сервера — Storage::url()

Если хост настроен верно, то «видны» только файлы, находящиеся в папке **storage/app/public** (если брать соединение по-умолчанию и драйвер - файловое хранилище).

Для доступу через веб нужно сформировать симлинк из папки **public** в папку **storage/app/public**:

```
php artisan storage:link
```

STORAGE: ГЕНЕРАЦИЯ URL

Кроме этого, Laravel для генерации url к файлам использует директиву **APP_URL**, которую нужно настроить в файле среды **.env**

Для удобства формирования url для файлов подключения по умолчанию, есть хелпер **url()**, в который передается путь к файлу, который передается в метод Storage::url().

ВАЛИДАЦИЯ В LARAVEL

В фреймворке имеются встроенные средства для валидации запроса.

При отправке запроса с неверными параметрами, выполняется редирект, и в сессии появляются сообщения об ошибке.

При этом в Laravel есть методы, позволяющие валидировать как наличие заполненного поля, так и учет правил заполнения полей.

Валидация в Laravel

Есть 2 варианта, как можно добавить валидацию в методе контроллера:

- Внутри метода
- Через отдельный объект типа Request

ВАЛИДАЦИЯ В LARAVEL ВНУТРИ МЕТОДА

Валидация внутри метода контроллера:

```
public function store(Request $request)
{
    $this->validate($request, [
        'title' =>
        'required|unique:posts|max:255',  'body'
        => 'required',
    ]);
}
```

title и body - это поля формы (в запросе). Значения массива - правила валидации

ВАЛИДАЦИЯ В LARAVEL: ОБЪЕКТ REQUEST

Для вынесения валидации в отдельный объект, нужно создать объект типа Request:

```
php artisan make:request
```

После создания, его нужно подключить в метод контроллера вместо Request:

```
public function store(PostRequest $request)
{


}
```

ВАЛИДАЦИЯ В LARAVEL: ОБЪЕКТ REQUEST

- файл запроса будет создан в папке **app/Http/Requests**
- 2 метода: **authorize()** и **rules()**
- при попытке пройти валидацию будет выдаваться ошибка:

This action is unauthorized.

- правила валидации указываются в методе rules;
- запрет авторизации запроса в методе authorize()

ВАЛИДАЦИЯ В LARAVEL: ПРАВИЛА ВАЛИДАЦИИ

Правила валидации указываются внутри массива, где ключом является поле, для которого должно действовать правило валидации, а значение - это сами правила валидации.

- Возможно указать сразу несколько правил.
- Правила указываются через |
- Параметры для правила указываются через :
- Значения для параметров правила перечисляются через запятую

ВАЛИДАЦИЯ В LARAVEL: ПРАВИЛА ВАЛИДАЦИИ

1. **between:max,min** — значение между min и max
2. **boolean** — значение **true, false, 1, 0, "1"**, или **"0"**.
3. **date** — валидная дата
4. **different:field** — значение, отличное от значения поля **field**.
5. **digits:value** — числовое поле, **value** — кол-во чисел.
6. **digits_between:min,max** — числовое поле, от **min** до **max** кол-во символов
7. **email** — валидный Email-адрес.
8. **integer** — целочисленное значение.
9. **json** — валидное JSON-значение.

ВАЛИДАЦИЯ В LARAVEL: ПРАВИЛА ВАЛИДАЦИИ

10. **max:value** — не более **value** символов
11. **min:value** — не менее **value** символов
12. **nullable** — может принимать значение null (поле не обязательное)
13. **numeric** — числовое значение
14. **required** — обязательно к заполнению
15. **size:value** — поле должно иметь **value** кол-вол символов
16. **same:field** — поле должно иметь такое же значение, как и поле **field**

Полный список правил, поддерживаемых Laravel можно найти в документации:

https://laravel.com/docs/5.6/validation#available-validation-rules

ВАЛИДАЦИЯ В LARAVEL: СООБЩЕНИЯ

Переопределение сообщений об ошибке:

- При валидации методом **validate()** внутри экшина, сообщения указываются массивом 3м параметром.
- При валидации в объекте Request — в нём же добавляется метод **messages()**, который возвращает массив сообщений об ошибках валидации.

ВАЛИДАЦИЯ В LARAVEL: СООБЩЕНИЯ

Сообщения об ошибках валидации указываются по схеме:

- ключ — название правила валидации
- значение — сообщение об ошибке

ВАЛИДАЦИЯ В LARAVEL: СООБЩЕНИЯ

В сообщении можно использовать

- :attribute — имя поля
- параметры из правила валидации — указать через двоеточие:

```
$messages = [
    'size'     => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
];
```

ВАЛИДАЦИЯ В LARAVEL: СООБЩЕНИЯ

При этом, можно для каждого поля указать свои сообщения. Для этого ключом нужно указать через точку название поля и правило:

```
$messages = [
    'name.size'=> 'The :attribute must be exactly :size.',
    'age.between' => 'The :attribute must be between :min -
    :max.',
];
```

ВАЛИДАЦИЯ В LARAVEL: ОТОБРАЖЕНИЕ

В представлении, если были получены ошибки валидации, появится $errors с информацией об ошибках валидации.

```
@if($errors->any())
        <div class="alert alert-danger">
                <ul>
                        @foreach ($errors->all() as $error)
                                <li>{{ $error
                        }}</li>  @endforeach

                </ul>
        </div>
@endif
```

ФОРМЫ И ЗАПОЛНЕННЫЕ ПОЛЯ И ХЕЛПЕР OLD

Данные из запроса можно получить через **old()**

Это решает проблему пропадания значений заполненных ранее полей.

Метод **old()** принадлежит объекту Request.

MIDDLEWARE

Middleware или «Посредники» — представляют из себя удобный механизм фильтрации запросов.

Кроме того, они позволяют выполнять различные действия со всем запросом (или ответом) в целом.

Возможно добавление своих Middleware.

MIDDLEWARE: ВСТРОЕННЫЕ

Middleware, проверяющий CSRF-токен в запросе:
\App\Http\Middleware\VerifyCsrfToken

Middleware убирающий лишние пробелы (trim):

\Illuminate\Foundation\Http\Middleware\TrimStrings

Middleware, приводящий пустые значения переменных в запросе к null:

\Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull

MIDDLEWARE: СОЗДАНИЕ

Создание Middlewaren:

```
php artisan make:middleware MyMiddleware
```

Создаст файл app/Http/Middleware/MyMiddleware.php с классом MyMiddleware
В файле единственный метод hanlde():

```
public function handle($request, Closure $next)

{
    return $next($request);
}
```

MIDDLEWARE: СОЗДАНИЕ

В методе handle() обрабатывается запрос.

После выполнения действий внутри middleware, будет продолжена обработка запроса.

Обработка непосредственно ответа:

```
public function handle($request, Closure $next)
{
    $response =
    $next($request);  return
    $response;
}
```

MIDDLEWARE: КАК ПОДКЛЮЧИТЬ К МАРШРУТУ

Для подключения middleware к маршруту в Laravel есть несколько способов:

1. Указать в маршруте:

```
Route::get('admin', ['uses' => 'Controller@index', 'middleware' =>
'auth']);  или

Route::get('admin', 'Controller@index')->middleware('auth');
```

MIDDLEWARE: КАК ПОДКЛЮЧИТЬ

2. Зарегистрировать для группы маршрутов:

В файле app\Http\Kernel.php указать в свойстве $middlewareGroups в нужной группе маршрутов

3. Зарегистрировать для всех маршрутов:
В файле app\Http\Kernel.php указать в свойстве $middleware

При этом можно указать как класс middleware, так и его алиас.

MIDDLEWARE: РЕГИСТРАЦИЯ

Для регистрации алиаса middleware — его следует зарегистрировать в файле app\Http\Kernel.php, добавив его в свойство $routeMiddleware как значение массива, где ключом будет его алиас.

ROUTE: ИМЕНОВАННЫЕ МАРШРУТЫ

Каждый маршрут может иметь своё имя. Задается оно следующим образом:

```
Route::get('about',
'PageController@about')->name('about');  или

Route::get('about', ['as' => 'about', 'uses' => 'PageController@about']);
```

ROUTE: ИМЕНОВАННЫЕ МАРШРУТЫ

Зачем нужны именованные маршруты?

Чтобы нигде жестко не привязывать никакие действия (ссылки, action формы, редиректы) на url.

ROUTE: ФОРМИРОВАНИЕ ССЫЛОК

Формируются ссылки по маршрутам хелпером **route():**

- 1-м аргументом передается имя маршрута
- 2-м аргументом передается массив параметров маршрута
- Если параметр только один — его можно передать без массива

//Route::get('about',

'PageController@about')->name('about'); route('about')

сформирует http://myhost/about

//Route::get('user/{id}',

'UsersController@profile')->name('user'); route('user', 12)

сформирует http://myhost/user/12

ROUTE: ФОРМИРОВАНИЕ ССЫЛОК

Если параметров несколько, они должны быть переданы в том же порядке, в котором зарегистрированы в маршруте. Названия ключей не имеют значения:

//Route::get('user/{id}/{action}', 'UsersController@action')->name('user_action');

route('user_action', [12, 'activate']) сформирует http://myhost/user/12/activate

ROUTE: СПИСОК МАРШРУТОВ

Вывод списка маршрутов в консоли:

```
php artisan route:list
```

ROUTE: РЕДИРЕКТ ПО МАРШРУТУ

Кроме формирования ссылок, маршруты можно использовать и при редиректах.

При формировании ответа, можно сделать редирект на нужную страницу:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;


class PagesController extends Controller
{
        public function aboutUs()

        {
                return redirect()->route('about');
        }
}
```

ROUTE: ПРОВЕРКА МАРШРУТА

Проверка маршрута по имени:

```
<?php

public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {


    }
    return $next($request);
}
//Здесь проверяется, что выполнение идёт на маршруте с именем profile
```

ROUTE: ГРУППИРОВКА ЗАПРОСОВ

Указание группе маршрутов посредника (Middleware):

```
Route::middleware(['auth'])->group(function() {
    Route::get('admin', 'Controller@index');
    Route::get('dashboard',
    'Controller@dashboard');
});
```

В методе middleware можно использовать массив из нескольких посредников.

ROUTE: ГРУППИРОВКА ЗАПРОСОВ

Указание Namespace для группы маршрутов:

```
Route::namespace('Auth')->group(function() {
    Route::get('admin', 'Controller@index');
    Route::get('dashboard',
    'Controller@dashboard');
});
```

И в данном случае контроллер Controller будет иметь namespace Auth (то есть Auth/Controller)

ROUTE: ГРУППИРОВКА ЗАПРОСОВ

Методы, позволяющие работать и с именами маршрутов (**name()**), и с url маршрутов (**prefix()**):

```
Route::prefix('panel')->group(function() {
    Route::get('admin',
    'Controller@index');
});
//данный маршрут будет доступен по пути /panel/admin
Route::name('panel.')->group(function() {
    Route::get('dashboard', 'Controller@dashboard')->name('dashboard');
});
//данный маршрут будет иметь имя panel.dashboard
```

ROUTE: ГРУППИРОВКА ЗАПРОСОВ

Методы группировки можно вкладывать друг в друга:

```
Route::prefix('panel')->group(function() {

    Route::name('panel.')->group(function() {

        Route::get('dashboard',
        'Controller@dashboard')->name('dashboard');

    });

});

//данный маршрут будет доступен по пути /panel/admin и по имени
panel.dashboard
```

ROUTE: ГРУППИРОВКА ЗАПРОСОВ

Метод **group()** позволяет указать сразу несколько условий:

```
Route::group([
        'namespace' =>
        'Admin',  'prefix' =>
        'panel',  'as'=>'panel'
    ], function() {
    Route::get('dashboard', 'Controller@dashboard')->name('dashboard');
});
```

- единственной особенностью в данном примере будет отличие индекса для префикса маршрута - вместо метода **name()** используется индекс **as**

ROUTE: ПОЛУЧЕНИЕ ИНФОРМАЦИИ

Также доступ к объекту маршрута можно получить через:

```
$route = Route::current();
```

Получить имя маршрута:

```
$routeName = Route::currentRouteName();
```

Получить метод (Controller@action) маршрута:

```
$routeAction = Route::currentRouteAction();
```

* не забудьте указать use Illuminate\Support\Facades\Route;

ROUTESERVICEPROVIDER

Вся организация работы роутера видна в данном провайдере:

1. Метод map() описывает маршруты
2. Методы mapApiRoutes и mapWebRoutes, которые запускаются в map(), описывают маршруты для всех “web” и “api” запросов.
3. В App\Http\Kernel в свойстве middlewareGroups по индексам web и api указаны подключаемые по группам маршрутов middleware

ROUTER В СТАРЫХ ВЕРСИЯХ LARAVEL

Основные изменения в модуле роутера произошли с приходом версии Laravel 5.3:

1. Файл с маршрутами назывался routes.php и лежал в app/Http/routes
2. Для группы маршрутов api были указаны middleware, но отдельного файла маршрутов не было