**theory Controller**

**--------------------------------------------------Dependency injection--------------------------------------------------**

**-Неявное внедрение зависимостей**(191)**:**

**Поиск по полю id:**

**Route:**

Route::get('/{id}', [Controller::class, 'methodName']);

**обычная форма Controller:**

public function methodName($id){

$context = ['arr' => ModelClassName::**find**($id)];

return view('pageName', $context);

}

**внедрение зависимостей Controller:**

public function methodName(**ModelClassName** $id) {

return view('pageName', ['arr' => $id]);

}

Для внедрения зависимостей необходимо соблюсти условия: в методе контроллера имя параметра $id должно совнадать с URL-параметром {id} маршрутизатора; в качестве типа парамньра должен быть указан класс Модели ModelClassName. Можно передавать

несколько параметров.

**Поиск по указанному столбцу таблицы можно реализовать несколькими способами:**

**первый способ -** добавлением имени столбца в URL-параметр:

**Route:**

Route::get('/{var**:columnName**}', [Controller::class, 'methodName']);

**Controller** - остается без изменений.

**второй способ -** переопределением метода модели возвращающего имя поля поиска:

**Внимание!!! В таком случае поиск всегда будет происходить по переопределенному полю!!!**

**Route и Controller**- остаются без изменений.

**Model:**

class Name extends Model {

public function **getRouteKeyName()**{

**return 'columnName'**;

}

}

**-Явное внедрение зависимостей**(192)**:**

Применяется при невозможности соблюдения усовий неявного внедрения зависимостей двумя способами: связи определенного URL-параметра и определенной модели; связи определенного URL-параметра и custom-ного поиска Модели. Реализуется а классе RouteServiceProvider в методе boot.

**-----------------------------------------------------------------------------------------------------------------------------------**

**- Разновидности контроллеров**(9.1)**:**

**- Контроллер функция**(9.1.1)**:**

Реализуется непосредственно в маршруте в виде анонимной функции.

Route:: get ( '/', function (){

return view('index');

)) ;

Route::get('/{bb}', function (App\Models\Bb $bb){

return view('detail', ['bb' => $bb]);

}) ;

**- Контроллер-класс**(9.1.2, 9.1.2.1, 9.1.2.3)**:**

php artisan make:controller NameController

--resource, --api, --invokable, --force, --mоdеl=<имя модели>, --раrеnt=<имя родительской модели>

**- Контроллер одного действия**(9.1.2.2)**:**

class IndexController extends Controller{

public function \_\_**invoke(){**

return view ( 'index');

}

}

**- Ресурсные и подчененные ресурсные контроллеры**(9.1.2.1)**:**

Имеют все необходимые CRUD методы для ресурсного и подчиненного ресурсного маршрута.

! я не уверен, но похоже, что метод **store** после выполнения логики "магией" перенаправляет на домашнюю страницу.

**- Получение объекта Request, обработка клиентских запросов**(9.3)**:**

Клиентский запрос представляется классом Illurninate\Http\Request. Получить объект этого класса можно одним из следующих способов:

**- средствами внедрения зависимостей:**

use Illurninate\Http\Request;

public function index(Request $request){

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

}

**- вызвав функцию request() без параметров:**

$request = request();

**- Из фасада Illurninate\Support\Facades\Request:**

use Illuminate\Support\Facades\Request;

$title = Request::input('title');

**- Извлечение данных из request(9.3.1):**

**Из POST, GET, URL-параметра:**

$request = **request()**; - получаем объект запроса со всеми параметрами POST, GET, URL.

$title = **$request->title**; - берем значение title

При обращении к свойству Laravel сначала ищет в запросе одноименный РОSТ-параметр, в случае неуспеха - одноименный GЕТ-параметр, далее - URL-параметр. Если поиски

подходящего значения не увенчаются успехом, возвращается null.

$title = request( 'title', ' '); - то же в одну строку

request(['title', 'content', 'address']) - массив значений

**При помощи методов Request:**

**route** (<имя ИRL-параметра> [, <значение по умолчанию>=null]) - возвращает значение URL-параметра с указанным именем, Если URL-параметр с заданным именем

отсутствует, возвращается значение по умолчанию;

**boolean**(<имя параметра>[, <значение по умолчанию>=fаlsе]) - возвращает true, если GET- или РОSТ-параметр с заданным именем хранит значение: 1, '1 ', true, 'true ', 'оn',' yes ', и false - в противном случае. Используется для проверки, был ли установлен присутствующий в веб форме флажок. Пример:

if ( $request->boolean ( 'agreed'){

// Флажок agreed установлен

}

**input()**; - возвращает ассоциативный массив со значениями всех GЕТ и РОSТ-параметров, что есть в запросе. Доступ к ним можно получить:

$allParams = Request::input();

$price = $allParams ['price'];

Можно указать значение по умолчанию которое верется в случает отсуцтвия свойства

input(<имя параметра> [, <значение по умолчанию>=null])

**query**(); - аналогичен input (), но работает только с GЕТ-параметрами.

**all()** - возвращает ассоциативный массив со значениями всех GЕТ и РОSТ-параметров, что есть в запросе.

**all**('title', 'content',) или **all**(['title', 'content']) - вернет массив указанных параметров.

**only**('title', 'content', 'address') - вернет массив указанных параметров.

**except**(['rubruc\_id', 'published']) или **except**('rubruc\_id', 'published') - вернет все

параметры кроме указанных.

Если параметр передается массивом, обычным или ассоциативным, для доступа к его элементам следует применять «точечную» нотацию:

$address0City = $request->input('addresses.O.city');

$address0Street = $request->input('addresses.O.street');

$address1City = $request->input('addresses.1.city');

**- Определение, присутствует ли в запросе нужное значение(9.3.2.):**

**has('name')** - есть ли в запроссе GET, POST параметр 'name', возвращает: есть - true, нет - false. Может принимать несколько параметров или массив.

if (Request::has('address') {

// Посетитель ввел адрес

}

**exists()** - то же самое, что и has () ;

**hasAny('name1', 'name2')** - возвращает true, если в текущем запросе есть хотя бы

один GЕТ или РОSТ-параметр с заданным именем;

**filled()** - аналогичен has (), но возвращает true только в том случае, если параметры с

заданными именами еще и не «пусты»;

**anyFilled()** - аналогичен hasAny (), но возвращает true только в том случае, если

параметры с заданными именами еще и не «пусты»;

**missing()** - возвращает true, если в текущем запросе отсутствуют GET- или РОSТ-

параметр (параметры) с заданным именем (именами);

**- Получение сведений о запросе(9.3.3):**

**secure()** - возвращает true, если текущий запрос был выполнен по протоколу HTTPS,

и false - если был выполнен по протоколу НТТР;

**ajах()** - возвращает true, если был выполнен АJАХ-запрос;

**pjax()** - возвращает true, если был выполнен РJАХ-запрос;

**expectsJson()** - возвращает true, если клиент запрашивает данные в формате JSON;

**prefetch()** - возвращает true, если текущий запрос является запросом на

предварительную загрузку файла;

**method()** - возвращает обозначение НТТР-метода, с помощью которого был

выполнен запрос;

**isMethod**( <обозначение НТТР-метода>) - возвращает true, если текущий запрос был

выполнен с помощью НТТР-метода с указанным обозначением

**isMethodSafe()** - возвращает true, если текущий запрос был выполнен с применением

НТТР-метода: GET, HEAD, OPTIONS или TRACE;

**path()** - возвращает путь, по которому был выполнен запрос, без начального и конечного слешей. Если был выполнен запрос к «корню)) сайта, будет возвращен символ

прямого слеша;

**is()** - возвращает true, если путь, по которому был выполнен запрос, совпадает с

одним из заданных шаблонов (смотри стр. 212);

**getScheme()** - возвращает обозначение протокола, по которому был выполнен запрос,

в виде строки 'http' или 'https';

**getHost ()** - возвращает интернет-адрес хоста без обозначения протокола и номера

порта (смотри стр. 213);

**getPort ()** - возвращает номер ТСР-порта, через который был выполнен текущий

запрос;

**getHttpHost()** - возвращает интернет-адрес хоста с указанием номера ТСР-порта,

если использовался нестандартный порт (смотри стр. 213);

**getSchemeAndHttpHost()** - возвращает полный интернет-адрес хоста, включающий

обозначение протокола;

**root()** - возвращает интернет-адрес «корня» сайта;

**url()** - возвращает интернет-адрес, по которому был выполнен текущий запрос, без

GЕТ-параметров;

**fullUrl()** - возвращает полный, с GЕТ-параметрами, интернет-адрес, по которому был

выполнен текущий запрос;

**fullUrlis()** - возвращает true, если полный интернет-адрес, по которому был выполнен

текущий запрос, совпадает с одним из заданных шаблонов (смотри стр. 213);

**routeis()** - возвращает true, если полученный клиентский запрос пришел через один

из маршрутов с заданными именами (смотри стр. 213);

**ip()** - возвращает IР-адрес клиента, выполнившего текущий запрос;

**ips()** - возвращает **массив** IР-адресов маршрутизаторов и сетевых шлюзов, через которые прошел текущий запрос. IР-адрес отправившего его клиента будет самым

последним в этом массиве;

**segment()** - возвращает сегмент пути с заданным номером, нумерация сегментов

начинается с 1 (смотри стр. 213);

**segments()** - возвращает массив со всеми сегментами пути;

**userAgent()** - возвращает значение заголовка User-Agent, возвращает строку:

Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/106.0.0.0 Safari/537.36;

**getPreferredFormat()** - возвращает строку с обозначением предпочтительного формата, извлеченным из текущего запроса (например, 'html '). Берется из первого

элемента списка, приведенного в заголовке Accept запроса;

**getAcceptableContentTypes()** - возвращает массив с обозначениями всех форматов

данных, поддерживаемых веб-обозреват;

**getLanguages()** - возвращает массив с обозначениями всех языков, поддерживаемых

веб-обозревателем;

**getPreferredLanguage()** - возвращает строку с обозначением предпочтительного языка;

**getCharsets()** - возвращает массив с обозначениями всех текстовых кодировок, поддерживаемых веб-обозревателем;

**getEncodings()** - возвращает массив с обозначениями всех форматов сжатия,

поддерживаемых веб-обозревателем;

**getProtocolVersion()** - возвращает строку с версией протокола НТТР в формате

НТТР/<номер версии>;

**headers** - все заголовки текущего запроса в виде объекта особого класса. Для получения значения заголовка с заданным именем следует вызвать у этого объекта метод

get ( <имя заголовка>) . Пример: $h = request()->headers->get('Cache-Control');

**server** - сведения о веб-сервере и среде исполнения, полученные из массива $\_SERVER,

в виде объекта аналогичного класса: $s = request()->server->get('SERVER\_SOFTWARE');

**- Генерирование интернет-адресов(9.4):**

**url()** - функция генерирует интернет-адрес протоколом НТТР или HTTPS и заданными параметрами;

$url = url('rubrics/2/3') или $url = url('rubrics', ['2', '3']) результат: http://localhost:8000/rubrics/2/3

$url = url('rubrics', ['2', 'З'], true) результат: https://localhost:8000/rubrics/2/3

Если параметру нттрs дать значение true, будет сгенерирован интернет-адрес, использующий протокол HTTPS, если дать значение false - интернет-адрес с протоколом

НТТР, если null - интернет-адрес с текущим протоколом.

$url = url()->**current()** - текущий интернет-адрес без GЕТ-параметров;

$url = url()->**full()** - текущий интернет-адрес с GЕТ-параметрами;

$url = url()->**previous()** - интернет-адрес предыдущей страницы;

**secure\_url()** - функция генерирует интернет-адрес протоколом HTTPS и заданными параметрами;

$url = secure\_url('rubrics', ['2', '3']) результат: https://localhost:8000/rubrics/2/3

**route()** - функция похожа на url(), но работает с именованными маршрутами:

$url = route('nameRoute', ['urlOptionName' => value]);

$url = route('rubric', ['rubric' => 2]);

Так-же route() поддерживат несколько интересных методав (см стр 216);

**action()** - функция принимает не имя маршрута, а имя контроллера и его действия;

$url = action([App\Http\Controllers\MainController::class, 'index']);

При вызове этой функции фреймворк просматривает список маршрутов и генерирует интернет-адрес на основе первого маршрута, в котором записаны указанные контроллер и действие и который имеет в шаблонном пути заданные в массиве ИRL-параметры.

**-----------------------------------------------Генерирование ответов-----------------------------------------------**

**- Ответы в виде объектов класса View(9.5.1.1):**

**view**() - генерирует объект класса View на основе заданного шаблона страницы и необходимого контекста.

public function show(modelClassName $modelObj){

**return view**('bladeName', ['arrKeyName' => $modelObj]);

}

Если view находится в папке путь можно указать: 'folder/bladeName' или 'folder.bladeName'

Класс View поддерживает следующие полезные методы:

**with**() - добавляет в контекст шаблона новые переменные:

$view = view('rubric');

$view->with('bbs', $rubric->bbs()->get() );

return $view->with(['rubric' => $rubric, 'title' => 'Рубрики']);

**toHtml**() - возвращает строку с НТМL-кодом страницы, сгенерированной на основе

текущего шаблона.

Объект класса Illurninate\View\Factory имеет несколько полезных методов

(формированию страницы, поиска шаблона, существования щаблона), сотри стр.218-219.

**- Ответы в виде объектов класса Response(9.5.1.2):**

**response**() - используется для генерации простых страничек. Рендеринг выполняется в момент генерирования ответа, а не перед его отправкой.

public function \_\_invoke(){

return response('this text will appear on the page');

}

Можно задать дополнительные параметры(см.стр.219, имхо лишнее).

**- Отображение файла в веб-обозревателе(9.5.2.1):**

use Illuminate\Support\Facades\Response;

return Response::**file**('images/pushpin-red.jpg');

имхо не очень нужная штука, картинку можно вставить непосредственно в blade, единственное есть прикольные функции удаления файла после отправки и передачи

запросов к картинки серверу для увеличения скорости обработки (см.стр.219-220).

**- Отправка данных в форматах JSON и JSONP(9.5.2.3) и "Пустой" ответ(9.5.2.5):**

имхо используется для API запроссов.

**- Дополнительные параметры Response(9.5.3):**

**header**() - добавляет в текущий ответ заголовок с указанными параметрами:

public function rubric(){

response('this text will appear on the page')->header('content-Type', 'text/plain');

}

**withНeaders**() - добавляетв текущий ответ заголовки из заданного массива

return response('Здecь будет перечень объявлений.')

->withHeaders(['Content-Type' => 'text/plain',

'Cache-Control' => 'no-cache']);

**setStatusCode**() - задает у текущего ответа код статуса:

return rеsроnsе('Объявление не найдено! ')->setStatusCode(404);

**setCharset**(<обозначение кодировки>) - указывает у текущего ответа текстовую

кодировку с заданным обозначением.

**- Перенаправления в контроллере(9.5.4):**

**redirect()** - перенаправит на выбранный маршрут, обычно после выполнения какой-то логики кондроллером. Можно выполнить перенаправление по протоколу HTTP или HTTPS,

по умолчанию протокол остается исходный.

**route()** и **action()** - смотри выше в конспекте и стр.224.

**redirect()->away()** - используется для перенаправления на внешние ресурся.

**redirect()->home(302)** - перенаправляет на маршрут '/home'.

**redirect()->refresh()** - метод должен обновить страницу, но браузер ругается на циклическое перенаправление. Поверхносный поиск в интернете ничего не дал, нужно будет

разобратся, т.к. метод очень полезный. Заменяется методом redirect()->back().

**redirect()->back()** - должен перенаправлять на предыдущую страницу, но работает

как "обновить страницу".

**- Ссылка-якорь(9.5.4):**

**back()->withFragment('Fragment')** - добавить якорь;

**back()->withoutFragment()** - убрать якорь.

**- Обработка ошибок(9.6):**

При необходимости сгенирировать ошибку для Laravel самостоятельно можно воспользоватся одним из способов:

**abort()** - отправляет страницу с сообщением об ошибке с заданным кодом;

**abort\_if()** - отправляет сообщение об ошибке с заданным кодом, если указанное условие истинно;

**abort\_unless()** - отправляет сообщение об ошибке с заданным кодом, если указанное условие ложно;

**---------------------------------------------------------Валидация---------------------------------------------------------**

**- Извлечение введенных данных(10.1):**

В параграфе даются примеры html кода разных форм ввода и методов для их извлечения из request: **поле ввода**, **флажок**, **раскрывающийся список**(с выбором одного или нескольких

элементов), **получение всех данных** из reqouest и **значения null**.

**- Неявное создание валидатора(10.2.1.1):**

Валидация производится вызовом у объекта запроса request метода validate(), которому необходимо передать ассоциотивные массивы с правилами валидации и сообщениями об ошибках. Массивы с правилами можно объявить как внутри метода, где выполняется валидация, так и в самом классе, если эти правила будут применятся в нескольких методах.

class ValidationController extends Controller{

private const VALID\_RULS = ['item'=>'max:30'];

private const VALID\_MESS = ['item.max'=>'item must be no longer than 30 characters'];

public function homePage(Request $request){

$validRuls = ['item' => 'required'];

$validMessages = ['item.required' => 'entry field cannot be empty'];

**$validated = $request->validate($validRuls, $validMessages);**

Если валидация не пройдена нас вернет на ту-же страничку с сообщениями об ощибке (их нужно обработать на html страничке, смотри книгу). Если валидация пройдена метод validated() вернет массив с данными для работаты контроллера (например сохранения в

БД).Далее пользователя нужно куда-то перенаправить.

**- Валидация с явным созданием валидатора(10.2.1.2):**

Имеет расширенный набор методов валидации и возможность изменять логику действия валидатора. ИМХО нужен только при необходимости внести изменения в логику действия валидатора, например выполнить кастомное действие при ошибки валидации.

Формат вызова:

use Illuminate\Support\Facades\Validator;

class ValidationController extends Controller{

...(тут как в примере выше правила и метод)

**$validator = Validator::make($request->all(), $validRuls, $validMessages);**

**$validator->validate();**

**- Валидация формальным запросом(10.2.2):**

Формальным запросом называют класс, самостоятельно выполняющий валидацию данных.

Создать формальный запросс:

php artisan make:request Name

Пример класса валидации:

class FormRequestValidation extends FormRequest{

//в методе должна происходить авторизация пользователя или вернуть true

public function authorize(){

return true;

}

//правила валидации

public function rules(){

return [ 'formReqValid' => 'max:5'];

}

//сообщения об ошибках

public function messages(){

return [ 'formReqValid.max' => 'message must be not longer than 5 characters'];

}

}

Внимание!!! **Валидация происходит при создании объекта**(даже если объект создается внедрением зависимостей), а не при вызовем метода. Пример метода контроллера:

public function formRequestValidation(**FormRequestValidation $formRequest**){

//save in DB all data $formRequest->all();

}

**- Валидация массивов элементов управления(10.2.1.3):**

Массивами на back обычно возвращаютяся флажки, списки и т.п. Их валидация необходима во избежания "взлома редактированием html кода".

**front:**

<form action="{{ route('validationHome') }}" method="POST">

@csrf

<select name="arrValid[]" size="3" multiple>

<option value='1'>one</option>

<option value='2'>two</option>

<option value='3'>three</option>

</select>

<input type="submit" value="Send">

</form>

**back:**

public function validationHome(Request $request){

$validRuls = ['arrValid' => 'array', 'arrValid.\*' => 'integer'];

$validMessages = ['arrValid' => 'list must be arrey', 'arrValid.\*' => 'arrey elements must be integer'];

$validator = validator($request->all(), $validRuls, $validMessages);

$validator->validate();

//сценарий успеха

}

**- Не сохраняемые в сессию GET и POST парраметры(10.2.1.2):**

Указываются в классе **App\Exeptions\Handler** в защищенном свойстве **$dontFlash**

class Handler extends ExceptionHandler{

protected $dontFlash = [

'current\_password',

'password',

'password\_confirmation',

];

}

**- Написание правил валидации(10.2.3):**

Правил валидации записывается в виде ассоциативного массива. Ключи элементов должны совпадать с именами GЕТ и РОSТ-параметров, хранящих введенные посетителем значения. Значения элементов зададут правила валидации:

$validationRules = ['content' => 'required']; - поле content должно быть заполненным.

Количество правил валидации может быть произвольным и должно разделятся "|":

$validationRules = ['item' => 'required|string'];

Многие правила валидации принимают параметры, которые записываются через двоеточие

после имени правила валидации и отделяются друг от друга запятыми:

$validationRules = ['title' => 'max:30']; - длинна title не должна привышать 30 символов

$validationRules = ['price' => 'digits\_between: 2, 50'];

Правила валидации также можно записать в виде массива, отдельный элемент которого

содержит отдельное правило:

$validation\_rules = ['title' => ['required', 'max:50']];

Правила валидации:

**required** - должно быть в запросе и не должно быть пустым;

**present** - должно быть в запросе;

**filled** - если есть в запросе должно быть заполненно или отсуцтвовать в запросе;

**sometimes** - если есть в запросе проходит валидацию;

'hobbies' => 'sometimeslrequired'

**string** - должно быть строкой, содержащей любые символы;

**alpha** - должно быть строкой, содержащей только буквы;

**alpha\_num** - должно быть строкой, содержащая только буквы и цифры;

**alpha\_dash** - должно быть строкой, содержащая только буквы, дефисы и символы подчеркивания;

**numeric** - целое или вещественное число;

**integer** - целое число;

**date** - значение даты в формате, поддерживаемом функцией РНР strtotime();

**timezone** - обозначение временной зоны из числа возвращаемых функцией РНР

timezone\_identifiers\_list();

**boolean** - логическая величина в формате true, false, 1, 0, применяется совместно с флажками;

**url -** интернет-адрес;

**active\_url** - реально существующий интернет-адрес (его существование проверяется);

**email:**параметры - адрес электронной почты. В параметре можно указать следующие валидаторы:

**rfc** - обычная проверка соответствия адреса стандартам ( его существование

в реальности не проверяется);

**strict** - аналогично rfc, но проверка более строгая;

**dns** - проверяет, существует ли почтовый сервер, присутствующий в заданном адресе, на самом деле;

**spoof** - проверяет, не присутствуют ли в адресе недопустимые символы;

**filter** - использует средства валидации адресов, встроенные в РНР.

Пример: 'user\_email' => 'email:rfc,dns,spoof'

**ip** - IР-адрес любого формата;

**ipv4** - IР-адрес формата IPv4;

**ipv6** - IР-адрес формата IPv6;

**uuid** - универсальный уникальный идентификатор (хз);

**json** - данные в формате JSON;

**array** - массив;

**nullable** - текущее значение может быть равно null;

**password:**"страж" - текущее значение должно совпадать с паролем текущего пользователя. Можно указать имя используемого стража (если не указан, будет использован страж по умолчанию, - подробно об этом в главе 13):

'retype\_your\_password\_to\_continue' => 'password:api'

**confirmed**- текущее значение должно совпадать со значением, имеющим имя формата <имя текущего значения>\_ confirmation (например, текущее значение

password должно совпадать со значением password\_confirmation);

**accepted** - одна из строк: 'yes ', 'on', '1' или 'true'. Обычно применяется с флажками вида: Я прочитал(а) пользовательское соглашение;

**required\_if:"other","value"**- текущее значение обязательно для ввода, если значение

с именем из параметра other равно величине value:

'hobbies list' => 'required\_if:has\_hobby,true' - см.расширенный пример в книге

**required\_unless:"other","values"** - текущее значение обязательно для ввода, если значение с именем из параметра other не равно ни одной величине из списка values (в котором отдельные величины разделяются запятыми):

'hobbies\_list' => 'required\_unless:has\_hobby,false,0,no'

**required\_with:"values"** - текущее значение обязательно для ввода, если было введено хотя бы одно из значений с именами, приведенными в списке values через запятую;

**required\_with\_all:"values"** - текущее значение обязательно для ввода, еслибыли введены все значения с именами, приведенными в списке values через запятую;

**required\_without:"values"** - текущее значение обязательно для ввода, если не было заполнено хотя бы одно из значений с именами, приведенными в списке values через запятую;

**required\_without\_all:"values"** - текущее значение обязательно для ввода, если не были заполнены все значения с именами, приведенными в списке values через запятую;

**same:"other"** - текущее значение должно совпадать со значением, имеющим имя из

параметра other:

required\_with\_all:<values>-'password\_confirm' => 'same:password'

**different:"other"** - текущее значение должно отличатся от значением, имеющим имя из параметра other;

**starts\_with:"values"** - строка, которая начинается на одну из подстрок, приведенных

в списке values через запятую:

'php\_variable\_name' => 'starts\_with:$'

**ends\_with:"values"** - строка, которая оканчивается на одну из подстрок, приведенных в списке values через запятую:

'executable\_file\_name' => 'ends\_with:com,exe,cmd,bat'

**in:"величины через запятую"** - текущее значение должно совпадать с одной из указанных величин:

'user\_role' => 'required | in:adrnin,editor,author' - см.расширенный пример в книге;

**nоt\_in:"величины через запятую"** - текущее значение не должно совпадать ни с одной из указанных величин. - см.расширенный пример в книге;

**digits:"digits"** - число, содержащее строго digits цифр:

'post\_index" => 'digits:6'

**digits\_between:"min","max"** - число, содержащее от min дотах цифр;

**date\_format:"format"** - значение даты в формате, заданном параметром format. Можно указать любой формат даты, поддерживаемый классом РНР DateTime. Нельзя

одновременно использовать с DATE. Пример:

'published\_at' => 'date\_format:d.m.Y'

**date\_equals:"date"** - значение даты, равное указанному в параметре date.В параметре можно задать либо строку в формате, поддерживаемом функцией РНР strtodate(), либо имя поля типа даты или временной отметки - тогда значение даты для

сравнения будет взято оттуда. Примеры:

'new\_year\_at' => 'dateldate\_equals:2020/01/01',

'signed\_at' => 'dateldate\_equals:given\_at'

**before:"date"** - значение даты более раннее, чем дата из параметра date ( поддерживает те же значения, что и параметр в правиле date\_equals):

'published\_at' => 'datelbefore:actual\_to'

**before\_or\_equal:"date"** - значение даты более раннее, чем дата из параметра date или равное ей;

**after:"date"** -значение даты более позднее, чем дата из параметра date:

'published\_at' => 'datelafter:created\_at'

**after\_or\_equal:"date"** - значение даты более позднее, чем дата из параметра date или равное ей;

**size:"value"** - правило «равно», поведение которого зависит от типа текущего

значения:

строка-должна иметь длину value символов;

число -должно быть равно величине value;

массив-должен иметь размер, равный value;

**min:"min"** - правило «не менее», поведение которого зависит от типа текущего значения:

строка -должна иметь длину не менее min символов;

число -должно быть не меньше величины min;

массив - должен иметь размер не менее min;

**max:"mах"** -правило «не более», поведение которого зависит от типа текущего начения:

строка - должна иметь длину не более max имволов;

число - должно быть не больше величины тах;

массив - должен иметь размер не более тах;

**between:"min", "mах"** - правило «между», поведение которого зависит от типа текущего значения:

строка -должна иметь длину от min до тах символов;

число -должно быть в диапазоне от min до тах включительно;

массив -должен иметь размер от min до тах;

**gt:"value"** - правило «больше», поведение которого зависит от типа текущего значения:

строка-должна иметь длину больше длины строки с именем value;

число -должно быть больше числа с именем value;

массив-должен иметь размер больше размера массива с именем value;

**gte:"value"** - правило "больше или равно", в остальном аналогично правилу gt;

**lt:"value"** -правило "меньше", в остальном аналогично правилу gt;

**lte:"value"** - правило "меньше или равно", в остальном аналогично правилу gt;

**regex: <регулярное выражение>** - текущее значение должно совпадать с заданным регулярным выражением. Последнее указывается в формате, поддерживаемом функцией РНР preg\_match();

**not\_regex: <регулярное выражение>** - текущее значение не должно совпадать с заданным регулярным выражением. Последнее указывается в формате, поддерживаемом функцией РНР preg\_ rnatch ();

**exists: <таблица> [, <поле> ]** - текущее значение должно существовать в заданном поле указанной таблицы. Если поле не указано, поиск значения будет выполняться в поле, чье имя совпадает с именем текущего значения. В параметре таблица можно указать:

имя таблицы -если она хранится в базе данных, используемой по умолчанию;

строку формата <имя базы данных>. <имя таблицы> -если требуемая таблица хранится в базе данных, отличной от используемой по умолчанию;

полный путь к классу модели, обслуживающей эту таблицу.

Примеры:

'usernarne' => 'exists:users'

'usernarne' => 'exists:App\\Models\\User'

'author\_narne'=>'exists:my\_users,username'

Сформировать правило можно программно, вызвав у фасада Rule метод exists () - см.стр.243

**unique: <табшща> [,<поле>]** - текущее значение не должно существовать в заданном поле указанной таблицы. Значения обоих параметров указываются в таком же формате, что и у правила exists. Пример:

'name' => 'requiredlunique:rubrics'

Сформировать правило можно программно, вызвав у фасада Rule метод exists () - см.стр.244

**distinct** - текущий массив должен содержать лишь уникальные значения:

'spares.\*.name' => 'distinct'

**in\_array: <other>** - текущее значение должно присутствовать в массиве с именем other:

'main title' => 'in\_array:titles.\*'

**exclude\_if:<other>,<value>**- текущее значение будет исключено из массива проверенных данных, возвращаемого методами validate () и validated (), если значение с именем из параметра other равно величине value;

**exclude\_unless:<other>,<value>** - тeкyщee значение будет исключено из массива проверенных данных, возвращаемого методами validate () и validated (), если значение с именем из параметра other не равно величине value;

**bail** - останавливает выполнение валидации, если одно из правил в заданной группе не выполняется. Используется только в составе групп правил. Пример:

'price' => 'requiredlnumericlbail'

**- Написание сообщений об ошибках ввода(10.2.4):**

Пишется в виде ассоциативного массива:

$validRuls = ['firstForm' => 'required|max:5',

'secondForm' => 'required|numeric'];

**$validMessages** = ['firstForm.required' => 'entry field cannot be empty'];

Ключем, обычно, служит GET/POST параметр, переданный из формы на странице, объединенный через точечную нотацию с одним или несколькими правилами.

Но ключ может состаять как из одного GET/POST параметра, так и из одного правила:

'required' => 'Обязательно занесите значение';

В синтаксис так-же входит **\*** - для указания любого элемента:

'spares.\*.count.integer' => 'Количество должно быть целым числом';

В значение элемента массива записывается сообщение возвращаемое на страницу при выполнении условий описанных в ключе элемента массива.

В тексте сообщения можно использовать литералы формата :<имя параметра правила валидации>. Вместо этого литерала будет подставлено значение параметра с указанным именем:

'name' => 'requiredlmax:40'

'name.max' => 'Дпина названия рубрики не должна превЬШiать :max символов'

Вместо литерала :attribute будет подставляться имя GET- или РОSТ-параметра (которое совпадает с наименованием соответствующего элемента управления):

'name.required' => 'Введите значение в поле :attribute'

Методы validate (), validateWithВag() контроллеров и метод make() фасада validator позволяют указать в последнем параметре ассоциативный массив с подстановками наименований. Ключи элементов такого массива должны соответствовать наименованиям элементов управления, а значения этих элементов будут выводиться в сообщениях об ошибках вместо литералов :attribute:

$validated = $request->validate($validation\_rules, $error\_messages,['title' => 'Название товара', 'price' => 'Цена товара']);

**- Извлечение ранее введенных данных на странице(10.2.5.):**

При некорректно введенных данных Класс/метод валидатора вернет предыдущую страницу с добавлением в серверную сессию ранее введенных данных. На странице их можно извлечь при помощи функции old():

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

Так же функция old() имеет параметр «значение по умолчанию» которое будет выводится при отсуцтвии старого значания:

<input name="title" value="{{ old('title'**, 'default value'**) }}">

Значение по умолчанию так же используется для вывода данных из DB на странице редактирования:

<input name="title" value="{{ old('title'**, $bb->title)** }}">

**- Извлечение сообщений об ошибках ввода в контроллере(10.2.6):**

При явном создании валидатора $validator = Validator::make() можно вызвать у него метод $validator->errors() который вернет объект класса Illuminate\Support\MessageBag с ошибками

валидации и имеющий методы работы с ними (См. Параграф).

**- Создание своих правил валидации(10.2.7):**

может быть осуществлено тремя способами:

**Правило анонимная функции(10.2.7.1):**

Правила валидации должны быть записаны в виде массива:

$validRuls = ['title'=>['required', 'max:50']];

Кастомное правило записывается как элемент массива в виде анонимной функции принимающий три параметра $name,$value, $fail (функция Laravel возвращающая сообщение об ошибке). В теле функции описывается логика:

use Illuminate\Support\Str;

$validRuls = ['title'=>['required',

function($name, $value, $fail){

//если не с большой буквы

if ($value != Str::ucfirst($value)) {

$fail('Error Message');

}

}

]

];

$validMessages = ['title.required' => 'entry field cannot be empty'];

$request->validate($validRuls, $validMessages);

Недостаток способа - функцию нельзя переиспользовать.

**Правила-расширения(10.2.7.2):**

Кастомное правило записывается в классе App\Providers\AppServiceProvider в теле метода

boot вызовом у фасада Validator метода extend():

use Illuminate\Support\Facades\Validator;

use Illuminate\Support\Str;

...

public function boot(){

Validator::extend('capitalized',

function($name, $value, $parameters, $validator){

return $value == Str::ucfirst($value);

},

'Capitalize the name (massege from AppServiceProvider methode)'

);

}

Методу extend передается: Имя правила, анонимная функция с логикой валидации, сообщение об ошибке по умолчанию.

Внимание!!! Имя правила должно быть отличным от стандартных правил валидации Laravel и без использования camelCase (не срабатывает, передаваемое из контроллера, сообщения об ошибке).

Анонимная функция должна принять четыре параметра и вернуть true или false.

В контроллере правило валидации используется как и стандартное правило валидации Laravel.

**Правила-объекты(10.2.7.3):**

Создаются в папке App\Rules командой:

php artisan make:rule ClassName

Имеет вид:

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

use Illuminate\Support\Str;

class CapitalizeClass implements Rule

{

public function \_\_construct(){ }

public function passes($attribute, $value){

return $value == Str::ucfirst($value);

}

public function message(){

return 'Capitalize the name (massege from Class)';

}

}

Метод passes() должен содержать логику валидации и возвращать true или false, метод message должен возвращать сообщение об ошибке (как передать в него сообщение из контроллера не разобрался).

Применение Класса в контроллере:

use App\Rules\CapitalizeClass;

$validRuls = ['validRulesObj'=>['required', new CapitalizeClass]];

$request->validate($validRuls);

**- Удаление начальных и конечных пробелов(10.3):**

По умолчанию у всех введенных посетителям данных удаляются начальные и конечные пробелы, кроме полей password и password\_confirmation.

Выполняется это при помощи класса посредника App\Http\Middleware\TrimStrings зарегистрированного в App\Http\Kernel.

При необходимости оставить начальные и конечные пробелы нужно в классе TrimStrings в свойстве $except в массив добавить наименование поля в котором не нужно удалять начальные и конечные пробелы.

Внимание!!! Удаления дублирующихся пробелов происходит где-то в процессе рендеринга

страницы или в браузере.

**--- Пагинация(12) ---** (https://laravel.com/docs/9.x/pagination)**:**

Пагинация - процесс разбиения списка на части.

В Laravel реализовано подсистемой "Пагинатор".

**- Автоматическое создание пагинатора(12.1):**

**Полнофункциональный пагинатор:**

Вычисляет общее количество выводимых частей, переходить на любую часть по ее номеру, предыдущую или следующую часть.

**paginate**(<количество записей в части>=15,

<массив с именами извлекаемых полей>=['\*'],

<имя GET- или РОSТ-параметра, содержащего номер текущей части>='раgе',

<номер извлекаемой чacти>=null)

примеры:

ModleName::paginate();

$bbsPaginated = Bb::latest()->paginate();

$bbsPaginated = Bb::latest()->paginate(5,['title', 'content', 'price'], 'page', 2);

**Упрощенный пагинатор:**

переход только на предыдущую или следующую часть

**simplePaginate**(формат вызова такой-же как у полнофункциогального)

**Вывод ссылок пагинатора на странице(12.1):**

У объекта пагинатора вызовом метода links(). В зависимости от формирования контекста, на странице вызывается по разному:

{{ $bbsPaginated->links() }}

{{ $context['simplePaginate']->links() }}

Примечание: по умолчанию у выводимых ссылок используется СSS-фреймворк Tailwind, в книге его меняют на Bootstrap. У меня криво работали оба (может я не правильно

использую СSS-фреймворки). Для исправления нужно извлечь views пагинатора из папки vendor (смотри конспект далее) и отредактировать.

У СSS-фреймворка Tailwind для нормализации размера стрелочек (выполненных векторной графикой SVG) ведущих на предыдущую и следующую страницы необходимо добавить свойства и изменить стандартное width="20" height="20" viewBox="0 0 20 10" (смотри

практику resources/views/vendor/pagination/tailwind.blade.php ~50 строка)

**- Дополнительные параметры пагинатора(12.2):**

Х.з. где их использовать, в Controller или View

**appends()** - добавляет к интернет-адресам гиперссылок пагинатора заданные GЕТ-параметры;

**withQuerystring()** - добавляет в интернет-адреса гиперссылок пагинатора все GЕТ-параметры, присутствующие в текущем интернет-адресе;

**fragment(**'якорь'**)** - добавляет в интернет-адреса гиперссылок пагинатора заданный якорь;

**withPath(**'путь'**)** - задает другой путь, используемый в гиперссылках пагинатора;

**onEachSize(**'количество гиперссылок'**)** - задает количество гиперссылок пагинатора,

выводящихся слева и справа от номера текущей части (не заработал).

**- Настройка отображения пагинатора(12.3):**

По умолчанию пагинатор выводится с применением шаблонов, являющихся частью фреймворка, находящихся в папке vendor без возможности их редактирования.

Для извлечения шаблонов необходимо выполнить команду:

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

все шаблоны переместятся в папку resources\views\vendor\pagination и будут доступны для редактирования.

**Внимание!** После выполнения команды Laravel, в дальнейшей работе, будет использовать

шаблоны для пагинатора из новой папки.

Список стандартных шаблонов пагинатора:

default.blade.php - шаблон полнофункционального пагинатора;

tailwind.blade.php - шаблон полнофункционального пагинатора, использующий СSS-фреймворк Tailwind (применяется по умолчанию при выводе полнофункционального пагинатора);

bootstrap-4(или 5).blade.php - шаблон полнофункционального пагинатора, использующий СSS-фреймворк Bootstrap;

semantic-ui.blade.php - шаблон полнофункционального пагинатора, использующий СSS-фреймворк Semantic UI (https://semantic-ui.com/);

simple-default.blade.php - шаблон упрощенного пагинатора;

simple-tailwind.blade.php - шаблон упрощенного пагинатора, использующий СSS-фреймворк Tailwind (применяется по умолчанию при выводе упрощенного пагинатора);

simple-bootstrap-4(или 5).blade.php - шаблон упрощенного пагинатора, использующий СSS-

фреймворк Bootstrap.

**Использование других CSS-фреймверков для пагинатора:**

в App\Providers\AppServiceProvider в теле его метода boot() вызвать статический метод

пагинатора указывающего необходимый CSS-фреймверк (нашел только Bootstrap):

use Illuminate\Pagination\Paginator;

class AppServiceProvider extends ServiceProvider{

public function boot(){

Paginator::useBootstrapFive();

или

Paginator::useBootstrapFour();

или другой CSS-фраймворк

}

}

**Использование своих шаблонов для пагинатора:**

для всего проекта

в App\Providers\AppServiceProvider в теле его метода boot() вызвать статический метод

пагинатора и передать ему путь к необходимой view:

use Illuminate\Pagination\Paginator;

class AppServiceProvider extends ServiceProvider{

public function boot(){

Paginator::**defaultView**('vendor.pagination.default'); - для полнофункционального

Paginator::**defaultSimpleView**('shared.my\_simple\_paginator'); - для упрощенного

}

}

для отдельной страницы

Во view этой страницы в методе links() указать путь к необходимой view пагинатора:

links('vendor.pagination.bootstrap-5')

**Использование методов и свойств пагинатора на странице:**

формируем контекст шаблона в контроллере

$context = ['fullPaginate'=>ModelName::paginate(5)];

return view('pagination.fullPaginate', ['context'=>$context]);

Доступ на странице к массиву хранящему гиперссылки пагинатора

$context['fullPaginate']->links()->**elements**

Доступ на странице к объекту пагинатора

$context['fullPaginate']->links()->**paginator**

Доступ на странице к методам объекта пагинатора

$context['fullPaginate']->links()->paginator->hasPages()

по такому полному пути происходит доступ к методам внутри views пагинатора (пригодится для кастомного пагинатора), но в контексте страницы уже присутствует объект пагинатора и можно обратится к его методам с тем-же результатом

$context['fullPaginate']->hasPages()

методы пагинатора(пригодится для кастомного пагинатора):

**hasPages()**- возвращает true, если в пагинаторе присутствует более одной части, и false- в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить пагинатор на экран;

**onFirstpage ()**- возвращает true, если текущая часть является первой, и false-в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить гиперссылку на предыдущую часть пагинатора;

**hasMorePages()** - возвращает true, если в пагинаторе присутствуют следующие части, и false- в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить гиперссылку на следующую часть пагинатора;

**currentPage()**- возвращает номер текущей части;

l**astpage()** - возвращает порядковый номер последней части (поддерживается только полнофункциональным пагинатором);

**previousPageUrl()**- возвращает интернет-адрес предыдущей части;

**nextPageUrl()** - возвращает интернет-адрес следующей части;

**count()** - возвращает реальное количество позиций в текущей части;

t**otal()** - возвращает полное количество позиций во всех частях пагинатора (поддерживается только полнофункциональным пагинатором);

**firstitem()** - возвращает порядковый номер первой позиции, присутствующей в текущей части;

**lastitem()** - возвращает порядковый номер последней позиции, присутствующей в текущей части;

**items()** - возвращает объект коллекции позиций(объекты модели), присутствующих в текущей части пагинатора;

**url**(<номер части>) - возвращает интернет-адрес части с указанным номером;

**getUrlRange**(<номер первой части>, <номер последней части>) - формирует интернет-адреса частей пагинатора с номерами, находящимися в указанном в параметрах метода диапазоне. В качестве результата возвращается ассоциативный массив, ключи элементов которого задают порядковые номера частей, а значения элементов - интернет-адреса соответствующих частей;

**perPage()** - возвращает максимальное количество позиций в части;

**getPageName()** - возвращает имя GET- или РОSТ-параметра, хранящего номер текущей части;

**setPageName**(<имя>) - задает новое имя для GET- или РОSТ-параметра, хранящего номер текущей части.

**- Создание пагинатора вручную(12.4):**

Штатные пагинаторы Laravel работают некорректно, если в построителе запросов была указана группировка или выборка ограниченного количества записей. Например вывод пяти самых "свежих" объявлений с пагинацией по одной записи в части. Тогда можно создать вручную полнофункциональный пагинатор конструктором класса Illuminate\Pagination\LengthAwarePaginator или упрощенный Illurninate\Pagination\Paginator передав им необходимые параметры (смотри примеры в книге).