GeekBrains

Laravel

Урок 7. Валидация данных в Laravel

Возможность Laravel валидировать данные, полученные из запроса пользователя.

Оглавление

Теория

- Правила валидации

Практика

- Создание валидации для добавления новой записи в таблицу новостей
- Вывод ошибок валидации

Русификация ошибок валидации

- Добавление локальных имен для названия элементов формы
- Тестирование формы

Практическое задание

Дополнительные материалы

Используемая литература

© geekbrains.ru

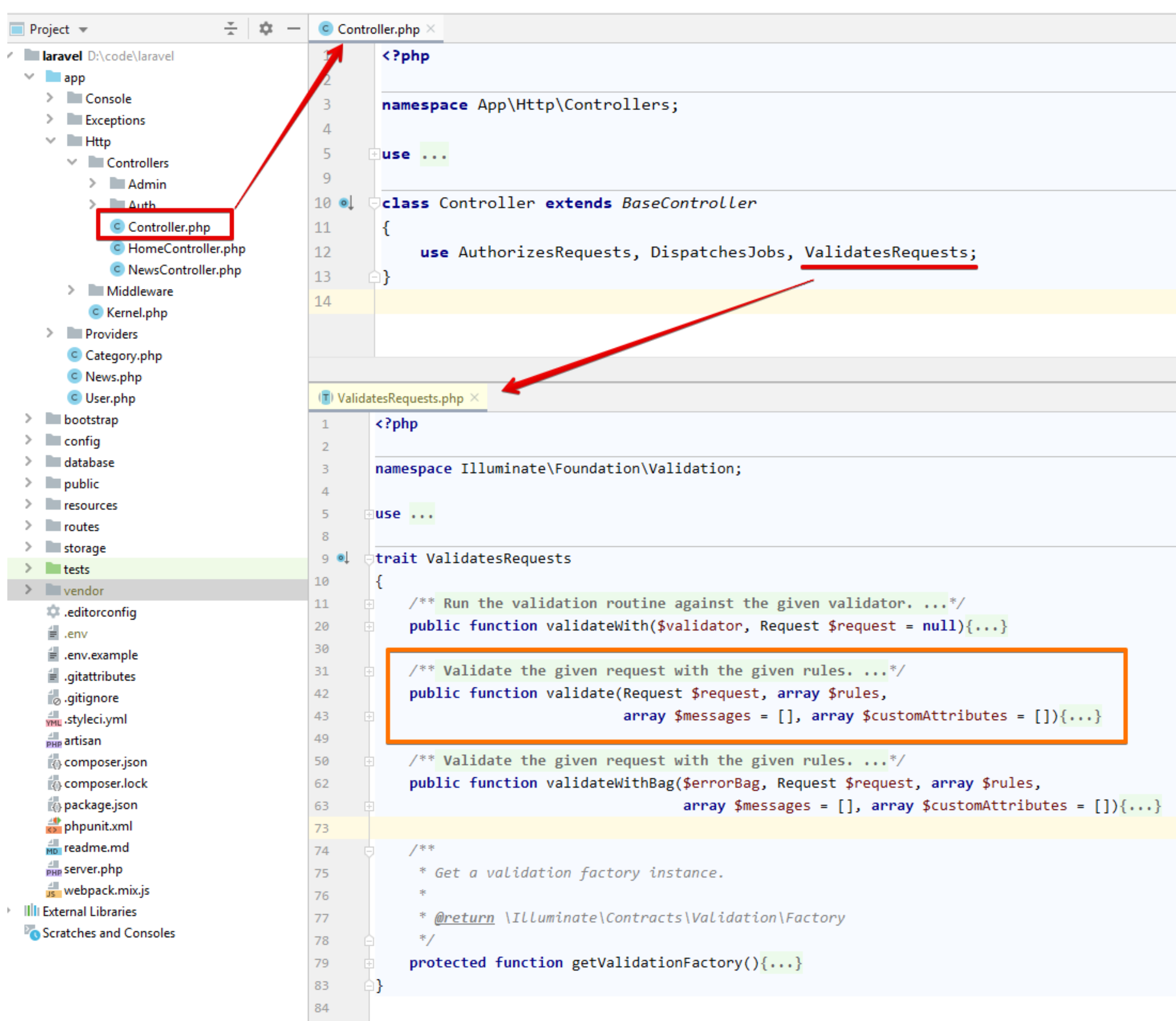
Теория

Правила валидации

Валидация — это проверка данных на соответствие заданным условиям и ограничением. Она выполняется с помощью правил валидации. Все правила, указанные в конкретной валидации, обязательны. При нарушении одного из правил валидация не состоится. Валидированию подвергаются данные, полученные из внешних источников. Все данные, которые приходят от пользователя, должны быть проверены — это защищает приложение от сбоев.

В Laravel есть несколько подходов для проверки данных. Базовый вариант — валидация данных в контроллере при помощи набора правил.

Базовый контроллер содержит трейт **ValidatesRequests**, в котором есть метод **validate**.



Этот метод принимает два обязательных параметра: объект класса **Request** и массив, в котором содержится набор правил валидации. Ключи этого массива соответствуют названиям элементов формы, ожидаемых в объекте **Request**. В значениях массива указывается строка, содержащая

правила валидации для соответствующего элемента. Правила разделяются в строке символом **|**. Если для правила необходимо передать значение — его указывают через символ **:**.

Когда валидация не проходит (одно или несколько правил не выполняются), то выбрасывается исключение. Если его не отловить, то данные об ошибках валидации будут сохранены во флеш-сообщения (те, которые после действий пользователей выводятся на экран только один раз). Тогда последует редирект на страницу, с которой был совершен запрос.

Пример массива с правилами валидации:

```
$rules = [
    'name' => 'required|max:10',
    'inform' => 'required|max:255',
    'user.email' => 'required|email:rfc,dns',
];
```

В данном примере указываются правила валидации для трех элементов формы:

- **name**,
- **inform**,
- **user[email]**.

Для элемента формы **email** указано два правила: он обязательно должен содержать значение, длина этого значения не должна превышать 10-символьную строку.

Рассмотрим некоторые правила валидации:

- **alpha** — правило проверяет, состоит ли указанное поле только из букв;
- **alpha_dash** — правило указывает, что поле должно содержать только буквенные и цифровые символы, а также тире и подчеркивания;
- **alpha_num** — правило указывает, что поле должно содержать только буквенные и цифровые символы;
- **date** — правило проверяет переданное значение на возможность его конвертации в дату;
- **email** — правило проверяет, что данные соответствуют валидному email;
- **integer** — правило проверяет, что передано целое число;
- **min:value** — проверяемое поле должно быть меньше или равно максимальному значению **value**. Для переданного числа проверяется соответствие указанному целочисленному значению, для строки — количеству переданных символов, для массива — переданному количеству элементов, для файла — количеству килобайтов;
- **max:value** — работает, как правило, **min**, но проверяет, что поле имеет минимальное значение **value**;
- **required** — правило указывает, что поле должно присутствовать во входных данных и не быть пустым;

- **unique:table,column** — правило проверяет, что поле не существует в таблице **table** в колонке **column**;
- **exists:table,column** — правило проверяет, что переданное значение содержится в поле **column** таблицы **table**.

Больше правил валидации на страницах документации Laravel: https://laravel.com/docs/5.8/validation#rule-size.

Практика

Создание валидации для добавления новой записи в таблицу новостей

Сначала добавим правила валидации для нашей модели. Для этого перейдем в класс **News** и создадим статический метод, который будет возвращать массив правил.

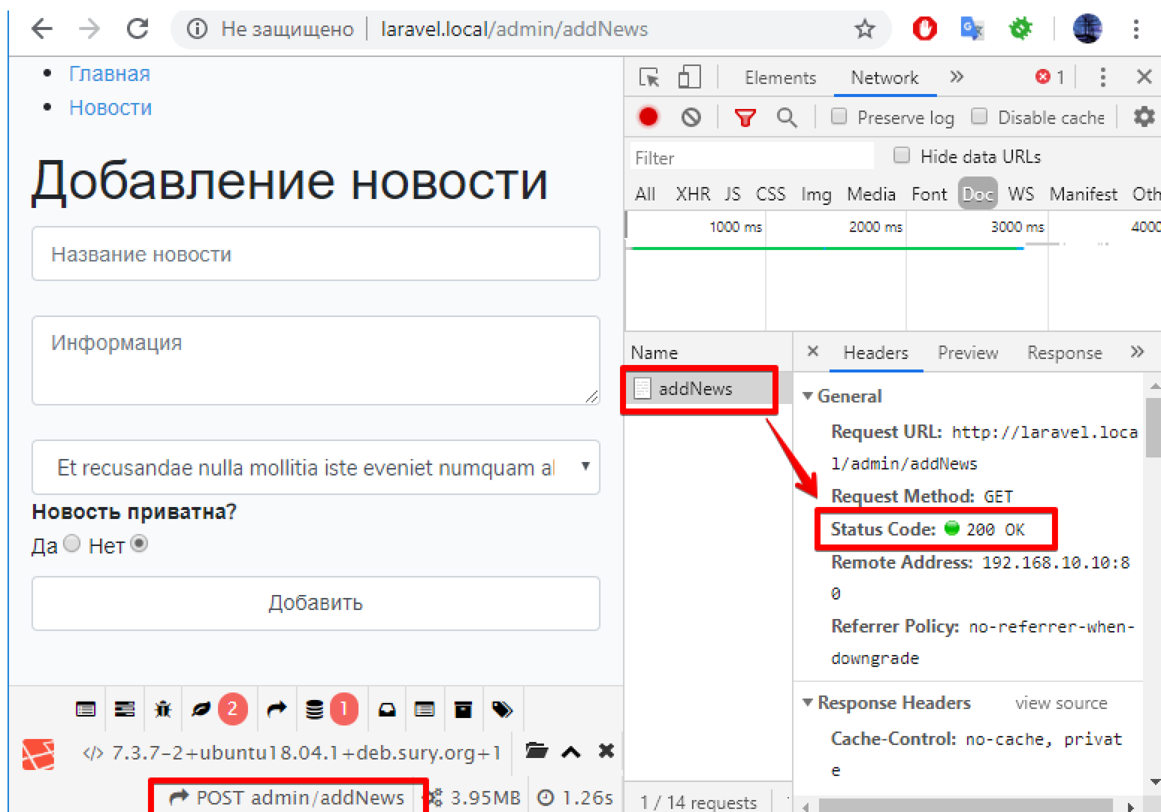
```php
public static function rules()
{
    $tableNameCategory = (new Category())->getTable();
    return [
        'title' => 'required|max:255',
        'inform' => 'required',
        'category_id' => "required|exists:{$tableNameCategory},id",
    ];
}
```

Рассмотрим подробнее эти правила. В них указаны поля, которые должны обязательно быть указаны при сохранении новой новости и при ее изменении. Кроме этого для поля **title** указано ограничение по количеству переданных символов — в нашем случае 255. Для поля **category_id** указано ограничение по id из таблицы, которая используется в модели **Category**. Имя таблицы вычисляется методом **getTable()** в соответствующей модели.

Чтобы правила начали работать, перейдем в **NewsController** в метод **add** и добавим в него вызов метода **validate** перед заполнением модели данными.

```php
public function add(Request $request)
{
    $news = new News();
    if ($request->isMethod( method: 'post')) {
        $this->validate($request, News::rules());
        $news->fill($request->all());
        $news->save();
        return redirect()->route( route: 'admin.news');
    }
    return view( view: 'admin.addNews', [
        'news' => $news,
        'rout' => 'admin.addNews',
        'title' => 'Добавление новости',
        'categories' => Category::query()->select(['id', 'title'])->get(),
    ]);
}
```

Теперь, если пользователь в нашем приложении допустит ошибки при заполнении формы, произойдет редирект на страницу заполнения данными. Но форма будет пустой, как будто пользователь ничего не вводил.

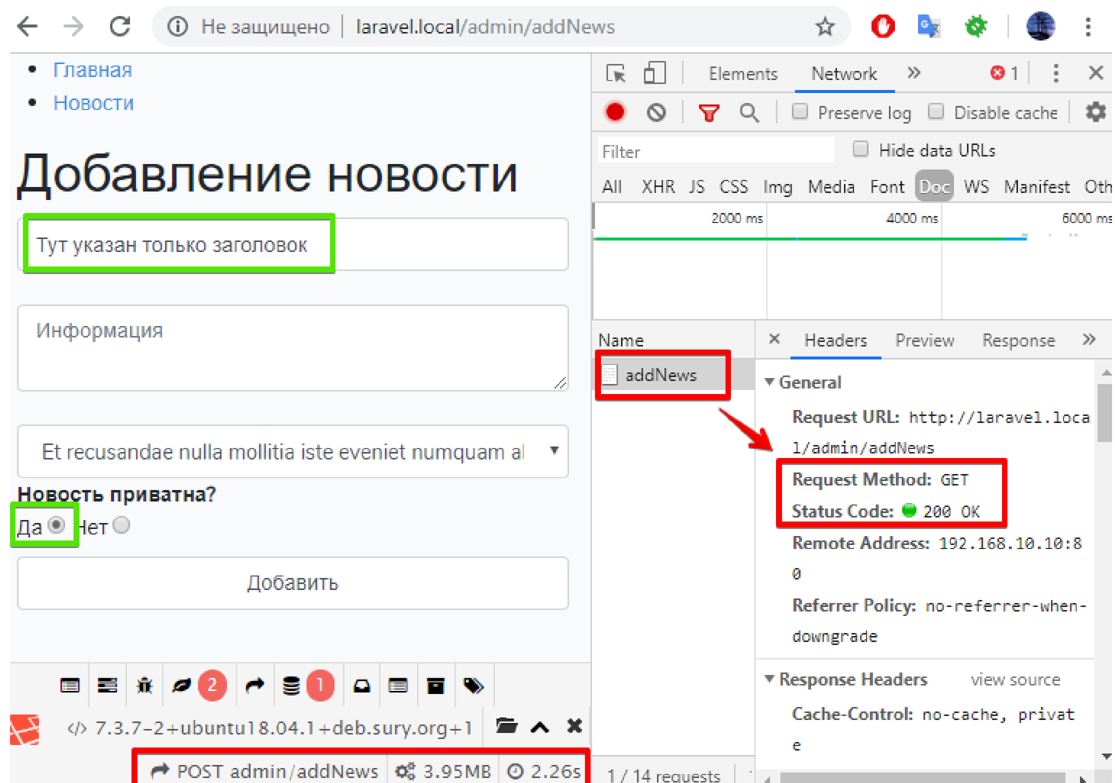


Такое поведение приложения не очень дружелюбно, поэтому добавим на форму те данные, которые были введены пользователем до отправки формы.

Данные, отправленные пользователем в форме, при возникновении ошибки валидации записываются в сессию в виде флеш-сообщений. Эти сообщения хранятся ровно до следующего запроса пользователя. Получить эти данные можно при помощи метода **old** объекта класса **Request**. Обновим метод для добавления данных:

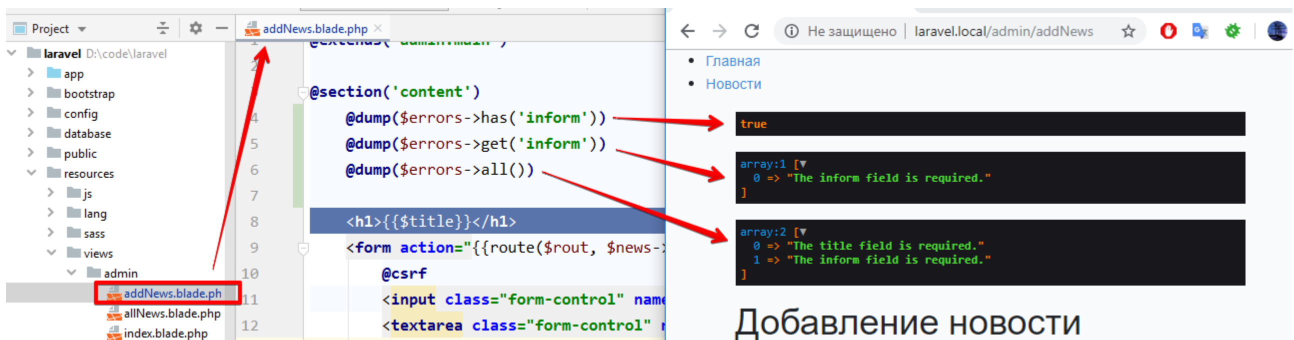
```php
    public function add(Request $request)
    {
        $news = new News();
        if ($request->isMethod( method: 'post')) {
            $this->validate($request, News::rules());
            $news->fill($request->all());
            $news->save();
            return redirect()->route( route: 'admin.news');
        }

        if (!empty($request->old())) {
            $news->fill($request->old());
        }

        return view( view: 'admin.addNews', [
            'news' => $news,
            'rout' => 'admin.addNews',
            'title' => 'Добавление новости',
            'categories' => Category::query()->select(['id', 'title'])->get(),
        ]);
    }
```

Таким образом получим следующий результат при ошибочно незаполненной форме:



Вывод ошибок валидации

Для вывода ошибок валидации воспользуемся переменной **$errors**, в которую при заполнении шаблона добавляется объект класса **Illuminate\Support\ViewErrorBag**, содержащий ошибки валидации.



Добавим красоты в вывод ошибок. Листинг кода **addNews.blade.php**:

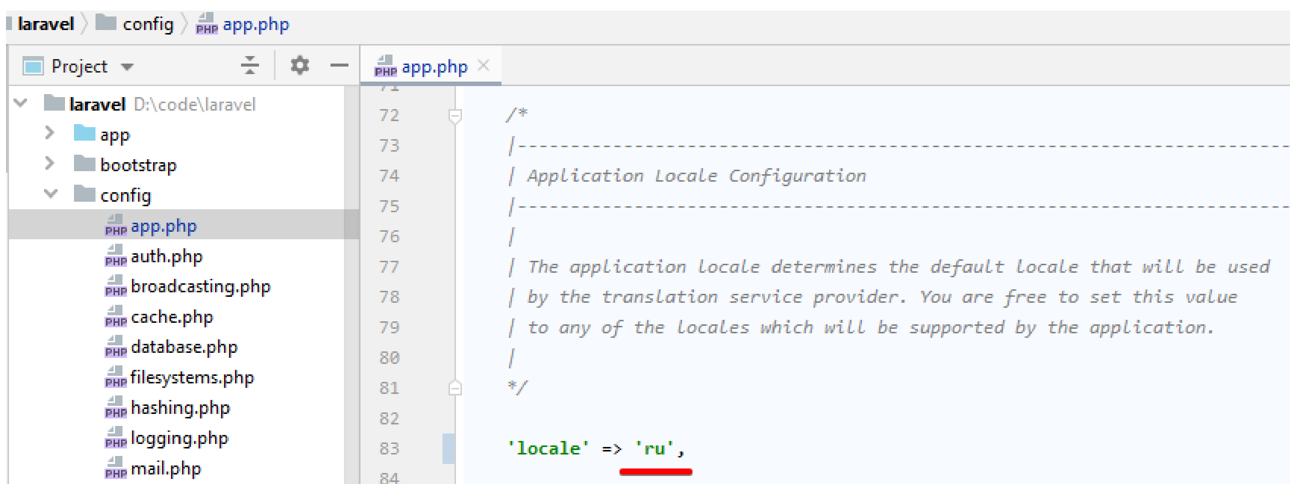
```
@extends('admin.main')

@section('content')
    <h1>{{$title}}</h1>
    <form action="{{route($rout, $news->id)}}" method="post">
        @csrf
        @if($errors->has('title'))
            <div class="alert alert-danger">
                    @foreach($errors->get('title') as $error)
                        <p style="margin-bottom: 0;">{{ $error }}</p>
                    @endforeach
            </div>
        @endif
        <input class="form-control" name="title" placeholder="Название новости"
value="{{$news->title}}"> <br>
        @if($errors->has('inform'))
            <div class="alert alert-danger">
                    @foreach($errors->get('inform') as $error)
                        <p style="margin-bottom: 0;">{{ $error }}</p>
                    @endforeach
            </div>
        @endif
        <textarea class="form-control" name="inform" placeholder="Информация">{{ $news->inform
}}</textarea> <br>

        @if($errors->has('category_id'))
            <div class="alert alert-danger">
                    @foreach($errors->get('category_id') as $error)
                        <p style="margin-bottom: 0;">{{ $error }}</p>
                    @endforeach
            </div>
        @endif
        <select name="category_id" class="form-control">
            @foreach($categories as $category)
                <option value="{{$category->id}}" {{$news->category_id == $category->id ?
'selected' : ''}}>
                    {{$category->title}}
                </option>
            @endforeach
        </select>

        <b>Новость приватна?</b> <br>
        <label>Да
            <input name="is_private" type="radio" value="1" @if ($news->is_private == 1) checked
@endif>
        </label>
        <label>Нет
```

```
            <input name="is_private" type="radio" value="0" @if ($news->is_private == 0) checked
@endif>
        </label> <br>
        <button class="form-control" type="submit">
            @if ($news->id) Изменить @else Добавить @endif
        </button>
    </form>
@endsection
```

Русификация ошибок валидации

Для русификации нужно указать приложению, что следует использовать русский язык по умолчанию, и добавить в папку с файлами языков русифицированные данные. Их можно скачать по ссылке: https://github.com/caouecs/Laravel-lang/tree/master/src/ru.

Изменение языка:



Добавленные файлы сообщений приложения на русском языке:

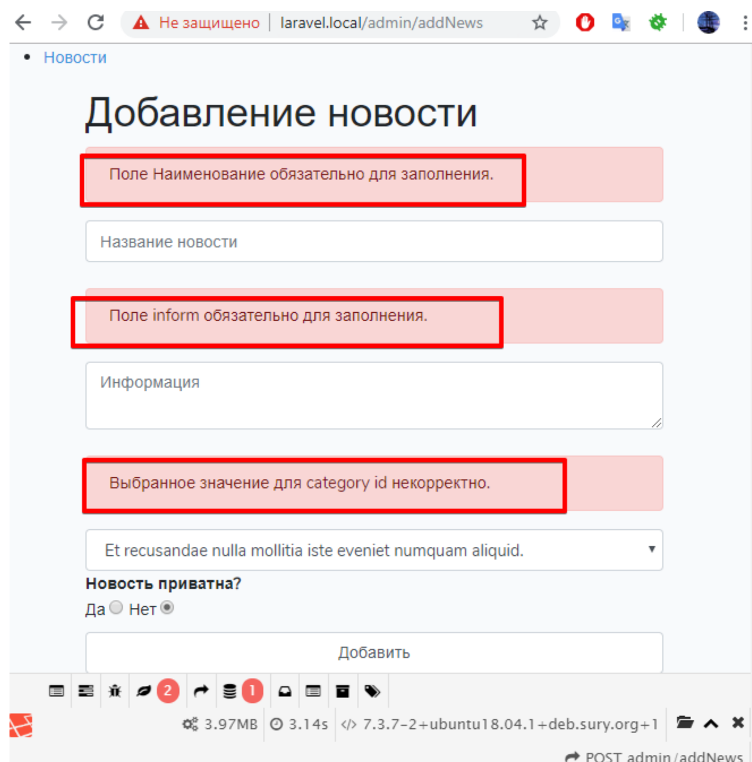
laravel 〉 resources 〉 lang 〉 ru 〉 validation.php

Project: laravel D:\code\laravel — app, bootstrap, config, database, public, resources (js, lang (en, ru: auth.php, pagination.php, passwords.php, validation.php), sass, views), routes, storage, tests, vendor, .editorconfig, .env, .env.example, .gitattributes, .gitignore, .styleci.yml, artisan, composer.json

validation.php | auth.php | pagination.php | passwords.php

```php
<?php

return [
    /*
    |--------------------------------------------------------------------------
    | Языковые ресурсы для проверки значений
    |--------------------------------------------------------------------------
    |
    | Последующие языковые строки содержат сообщения по-умолчанию, используемые
    | классом, проверяющим значения (валидатором). Некоторые из правил имеют
    | несколько версий, например, size. Вы можете поменять их на любые
    | другие, которые лучше подходят для вашего приложения.
    |
    */

    'accepted'             => 'Вы должны принять :attribute.',
    'active_url'           => 'Поле :attribute содержит недействительный URL.',
    'after'                => 'В поле :attribute должна быть дата после :date.',
    'after_or_equal'       => 'В поле :attribute должна быть дата после или равняться :dat
    'alpha'                => 'Поле :attribute может содержать только буквы.',
    'alpha_dash'           => 'Поле :attribute может содержать только буквы, цифры, дефис
    'alpha_num'            => 'Поле :attribute может содержать только буквы и цифры.',
    'array'                => 'Поле :attribute должно быть массивом.',
    'before'               => 'В поле :attribute должна быть дата до :date.',
    'before_or_equal'      => 'В поле :attribute должна быть дата до или равняться :date.'
    'between'              => [
        'numeric' => 'Поле :attribute должно быть между :min и :max.',
        'file'    => 'Размер файла в поле :attribute должен быть между :min и :max Килобай
        'string'  => 'Количество символов в поле :attribute должно быть между :min и :max.
```

Результат:



В сообщениях указываются названия элементов формы — это не информативно для пользователя. Также обратим внимание на то, что **title** было заменено на «Наименование».

Рассмотрим подробнее файл с переводом на русский язык сообщений валидации:

```
92      ],
93      'not_in'                => 'Выбранное значение для :attribute ошибочно.',
94      'not_regex'             => 'Выбранный формат для :attribute ошибочный.',
95      'numeric'               => 'Поле :attribute должно быть числом.',
96      'present'               => 'Поле :attribute должно присутствовать.',
97      'regex'                 => 'Поле :attribute имеет ошибочный формат.',
98      'required'              => 'Поле :attribute обязательно для заполнения.',
99      'required_if'           => 'Поле :attribute обязательно для заполнения, когда :other равно :value.',
100     'required_unless'       => 'Поле :attribute обязательно для заполнения, когда :other не равно :values.',
101     'required_with'         => 'Поле :attribute обязательно для заполнения, когда :values указано.',
102     'required_with_all'     => 'Поле :attribute обязательно для заполнения, когда :values указано.',
103     'required_without'      => 'Поле :attribute обязательно для заполнения, когда :values не указано.',
104     'required_without_all'  => 'Поле :attribute обязательно для заполнения, когда ни одно из :values не указано.',
105     'same'                  => 'Значения полей :attribute и :other должны совпадать.',
106     'size'                  => [
107         'numeric' => 'Поле :attribute должно быть равным :size.',
108         'file'    => 'Размер файла в поле :attribute должен быть равен :size Килобайт(а).',
109         'string'  => 'Количество символов в поле :attribute должно быть равным :size.',
110         'array'   => 'Количество элементов в поле :attribute должно быть равным :size.',
```

В каждом сообщении для каждого правила содержится плейсхолдер **:attribute** — метка в тексте, которая будет заменена при добавлении сообщения. На это место подставляется название элемента формы.

Но почему тогда вместо title указано «Наименование»?

Ответ можно найти в этом же файле:

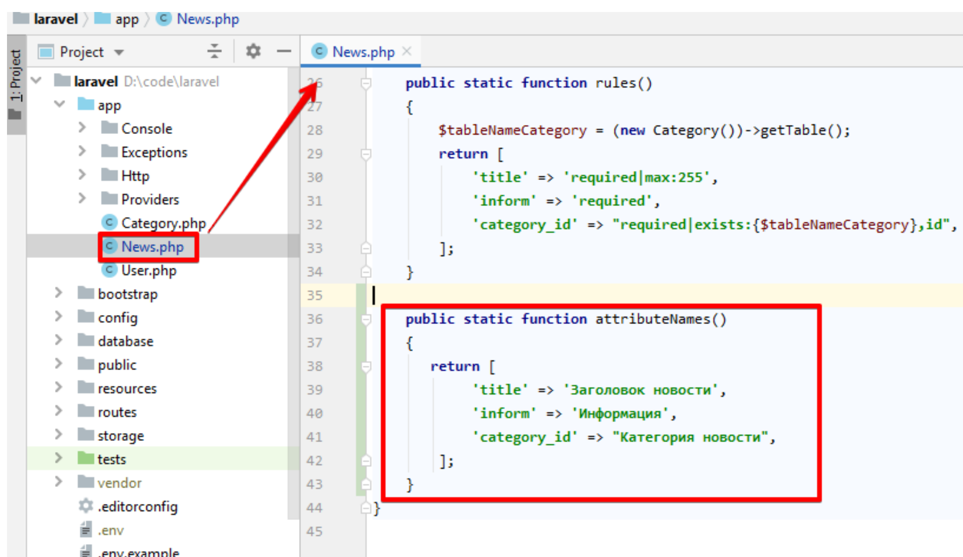
```
145     /*
146     |--------------------------------------------------------------------------
147     | Собственные названия атрибутов
148     |--------------------------------------------------------------------------
149     |
150     | Последующие строки используются для подмены программных имен элементов
151     | пользовательского интерфейса на удобочитаемые. Например, вместо имени
152     | поля "email" в сообщениях будет выводиться "электронный адрес".
153     |
154     | Пример использования
155     |
156     |   'attributes' => [
157     |       'email' => 'электронный адрес',
158     |   ],
159     |
160     */
161
162     'attributes' => [
163         'title'                 => 'Наименование',
164         'name'                  => 'Имя',
165         'username'              => 'Никнейм',
166         'email'                 => 'E-Mail адрес',
167         'first_name'            => 'Имя',
168         'last_name'             => 'Фамилия',
```

Таким образом, добавляя другие данные в указанный массив, можно подставлять нужные имена вместо названий элементов.

Но добавление данной информации в массив **attributes** приведет к глобальной подстановке.

Добавление локальных имен для названия элементов формы

Добавим еще один статический метод в модель **News**, который будет содержать в качестве ключей названия элементов формы добавления новости, а значений — их правильные имена.



Как этот метод использовать?

Еще раз внимательно рассмотрим метод для валидации данных. Он может принимать два необязательных параметра: **$messages** и **$customAttributes**.

```
public function validate(Request $request, array $rules,
                         array $messages = [], array $customAttributes = [])
{
    return $this->getValidationFactory()->make(
        $request->all(), $rules, $messages, $customAttributes
    )->validate();
}
```

В **$messages** можно передать сообщения, которые должны возвращать ошибки валидации. Поскольку нас вполне устраивают сообщения из добавленного перевода, будем в него передавать пустой массив. Сейчас нас интересует **$customAttributes**. В него передадим наш массив, который вернется из метода **attributeNames**.

```php
    public function add(Request $request)
    {
        $news = new News();
        if ($request->isMethod( method: 'post')) {
            $this->validate($request, News::rules(), [], News::attributeNames());
            $news->fill($request->all());
            $news->save();
            return redirect()->route( route: 'admin.news');
        }

        if (!empty($request->old())) {
            $news->fill($request->old());
        }

        return view( view: 'admin.addNews', [
            'news' => $news,
            'rout' => 'admin.addNews',
            'title' => 'Добавление новости',
            'categories' => Category::query()->select(['id', 'title'])->get(),
        ]);
    }
```

Результат:

Добавление новости

Поле Заголовок новости обязательно для заполнения.

Название новости

Поле Информация обязательно для заполнения.

Информация

Выбранное значение для Категория новости некорректно.

Et recusandae nulla mollitia iste eveniet numquam aliquid.

Тестирование формы

Для проведения автоматического тестирования форм установим специальный пакет **Laravel Dusk**. Он позволяет проводить функциональное тестирование приложения Laravel, имитируя нажатие кнопок, ссылок и другие действия.

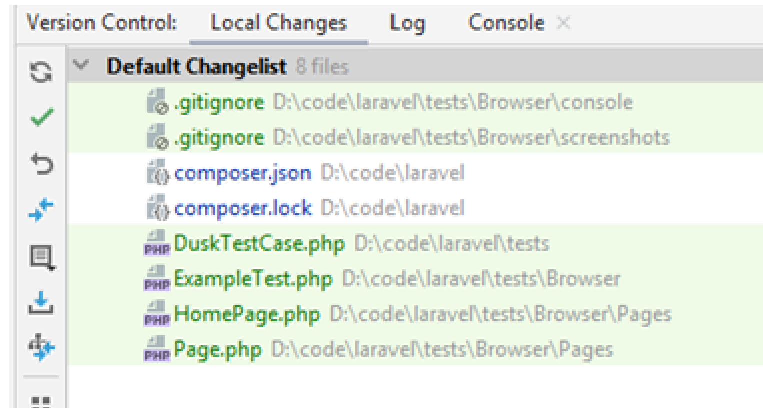
Подробная документация по пакету: https://laravel.com/docs/6.x/dusk.

Выполним следующие команду **composer require --dev laravel/dusk**. Она подгрузит в наш проект необходимые зависимости.

Далее выполним команду **php artisan dusk:install**.

```
vagrant@homestead: ~/code/laravel
vagrant@homestead:~/code/laravel$ php artisan dusk:install
Dusk scaffolding installed successfully.
Downloading ChromeDriver binaries...
ChromeDriver binaries successfully installed for version 78.0.3904.70.
vagrant@homestead:~/code/laravel$ |
```

В проекте появились файлы с примерами тестов.



Но прежде чем запустить тесты, надо установить браузерный движок. Для этого выполним команды:

- **wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub | sudo apt-key add -`**
- **‘sudo sh -c 'echo "deb [arch=amd64] http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sources.list.d/google-chrome.list''**
- **`sudo apt-get update && sudo apt-get install -y google-chrome-stable`**
- **`sudo apt-get install -y xvfb`**

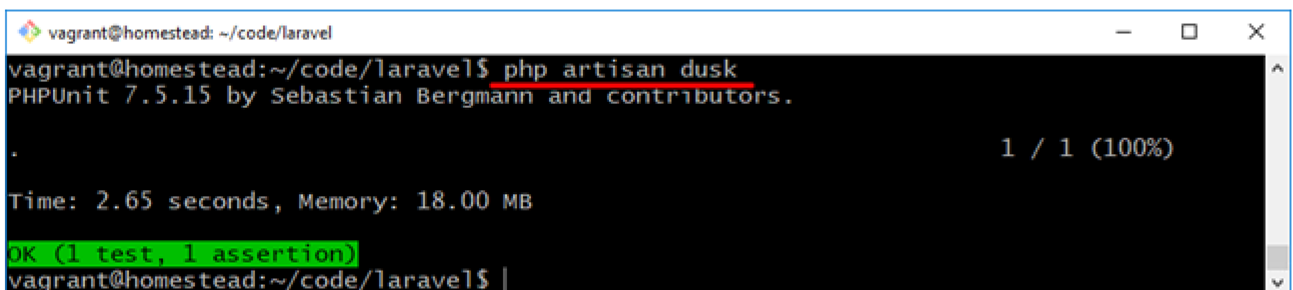
После этого установим соответствующий драйвер для нашего пакета: **php artisan dusk:chrome-driver**

Чтобы избежать ошибок с недостаточными правами для выполнения тестирования, изменим права в папке со скачанным пакетом: **chmod -R 755 vendor/laravel/dusk/bin/**

Почти готово. Подкорректируем файл примера теста, указав, что на странице точно будет другой текст:

```php
<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{

    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit( url: '/')
                    ->assertSee( text: 'Приветствие пользователя');
        });
    }
}
```

Запустить тесты можно так же, как и ранее в PhpStorm, или командой **php artisan dusk**.



Отлично! Тесты работают.

Удалим файл с пробным тестом **ExampleTest.php** и добавим новый: **php artisan dusk:make AddNewsTest**.

Протестируем два случая добавления новой новости: отправку правильно заполненной формы и формы, содержащей ошибки.

```php
<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;

class AddNewsTest extends DuskTestCase
{

    /**
     * @throws \Throwable
     */
    public function test1Example()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/admin/addNews')
                ->type('title', 'test')
                ->type('inform', 'informTest')
                ->press('Добавить')
                ->assertPathIs('/admin/news');
        });
    }

    /**
     * @throws \Throwable
     */
    public function test2Example()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/admin/addNews')
                ->type('title', '')
                ->type('inform', 'informTest')
                ->press('Добавить')
                ->assertSee('Поле Заголовок новости обязательно для заполнения.')
                ->assertPathIs('/admin/addNews');
        });
    }
}
```

Перед запуском теста рассмотрим подробнее, что написано в его данных. Происходит обращение к методу **browse**, которому в качестве параметра передается анонимная функция. Она принимает экземпляр класса **Laravel\Dusk\Browser**. Он содержит много методов для фейковых действий пользователя и для проверки результатов после их выполнения. У методов интуитивно понятные названия.

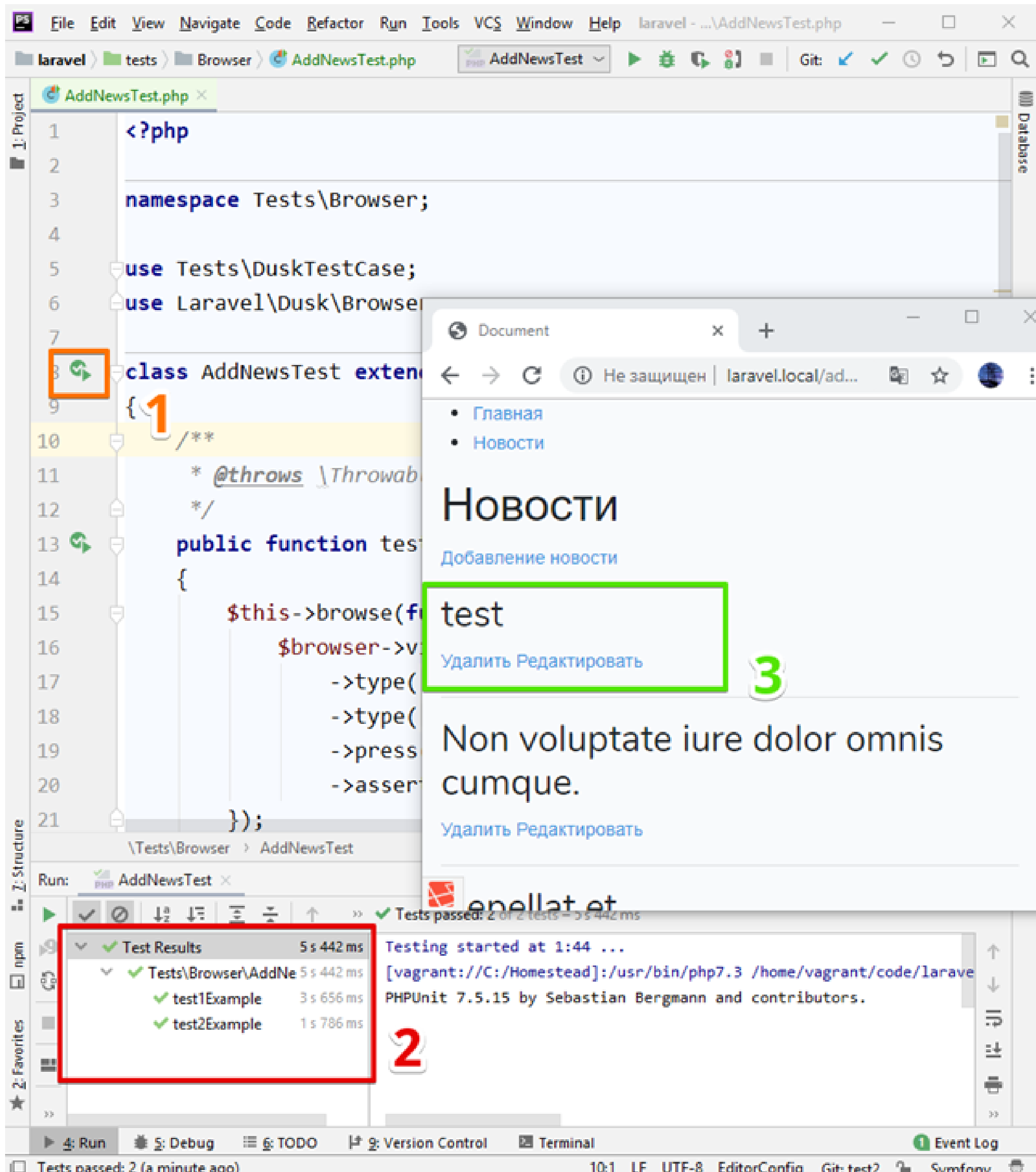
В тестах используются следующие методы класса **Laravel\Dusk\Browser**:

- **visit** — открытие указанного адреса приложения;
- **type** — ввод данных в элемент формы. Принимает два обязательных параметра: имя поля и значение для ввода;
- **press** — выполняет нажатие на кнопку с указанным именем;
- **assertPathIs** — проверяет, что пользователь после отправки формы окажется на указанном адресе приложения. Он находится не в самом классе **Laravel\Dusk\Browser**, а в трейте **Laravel\Dusk\Concerns\MakesUrlAssertions**, который используется в этом классе.

- **assertSee** — проверяет, что на странице есть указанный текст. Также используется из трейта **Laravel\Dusk\Concerns\MakesUrlAssertions,** подключенного в классе **Laravel\Dusk\Browser**.

Больше информации тут: https://laravel.com/docs/6.x/dusk.

Теперь запустим тест — для этого воспользуемся **phpStorm**.



Тест успешно выполнен, но появилась неожиданная запись в базе, которую не хотелось бы видеть по окончании тестов. Чтобы этого не происходило, укажем, что при проведении данных тестов следует использовать другую базу данных — в которую перед запуском самого теста будут выполнены все миграции и посевы данных.

Laravel позволяет использовать **sqlite** — базу данных, хранимую в файле.

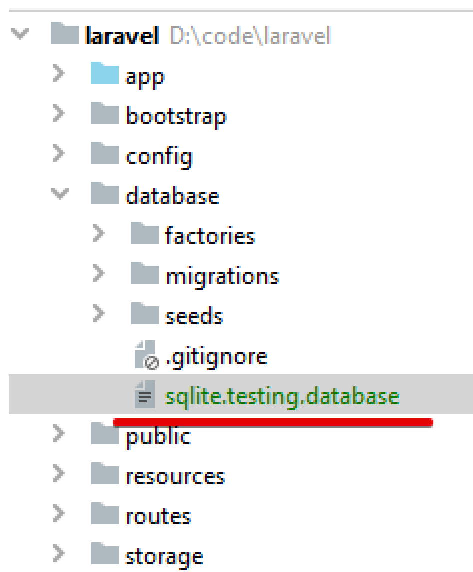
Сначала надо добавить новую конфигурацию с драйвером **sqlite**.

```
'default' => env( key: 'DB_CONNECTION', default: 'mysql'),

/*...*/

'connections' => [

    'sqlite' => [...],

    'sqlite_testing' => [
        'driver'   => 'sqlite',
        'database' => database_path( path: 'sqlite.testing.database'),
        'prefix'   => '',
    ],

    'mysql' => [...],
```

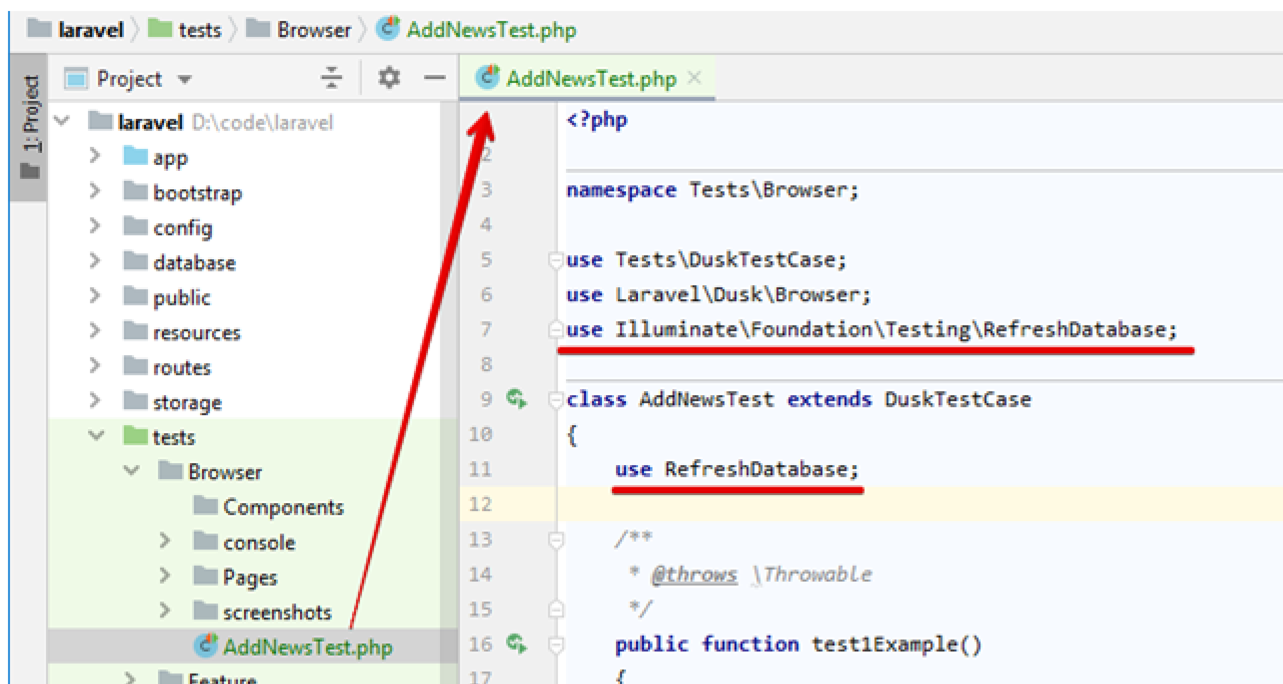
Далее создадим новый файл **env.dusk**, перенесем в него некоторые данные из основного env-файла (1) и укажем, что следует использовать конфигурацию базы, созданную ранее (2).

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:z+vMfuz98ufvxKiDq/qpIwBTVysxgRMelkUz/4BUDQY=
APP_DEBUG=true
APP_URL=http://laravel.local

LOG_CHANNEL=stack

DB_CONNECTION=sqlite_testing
```

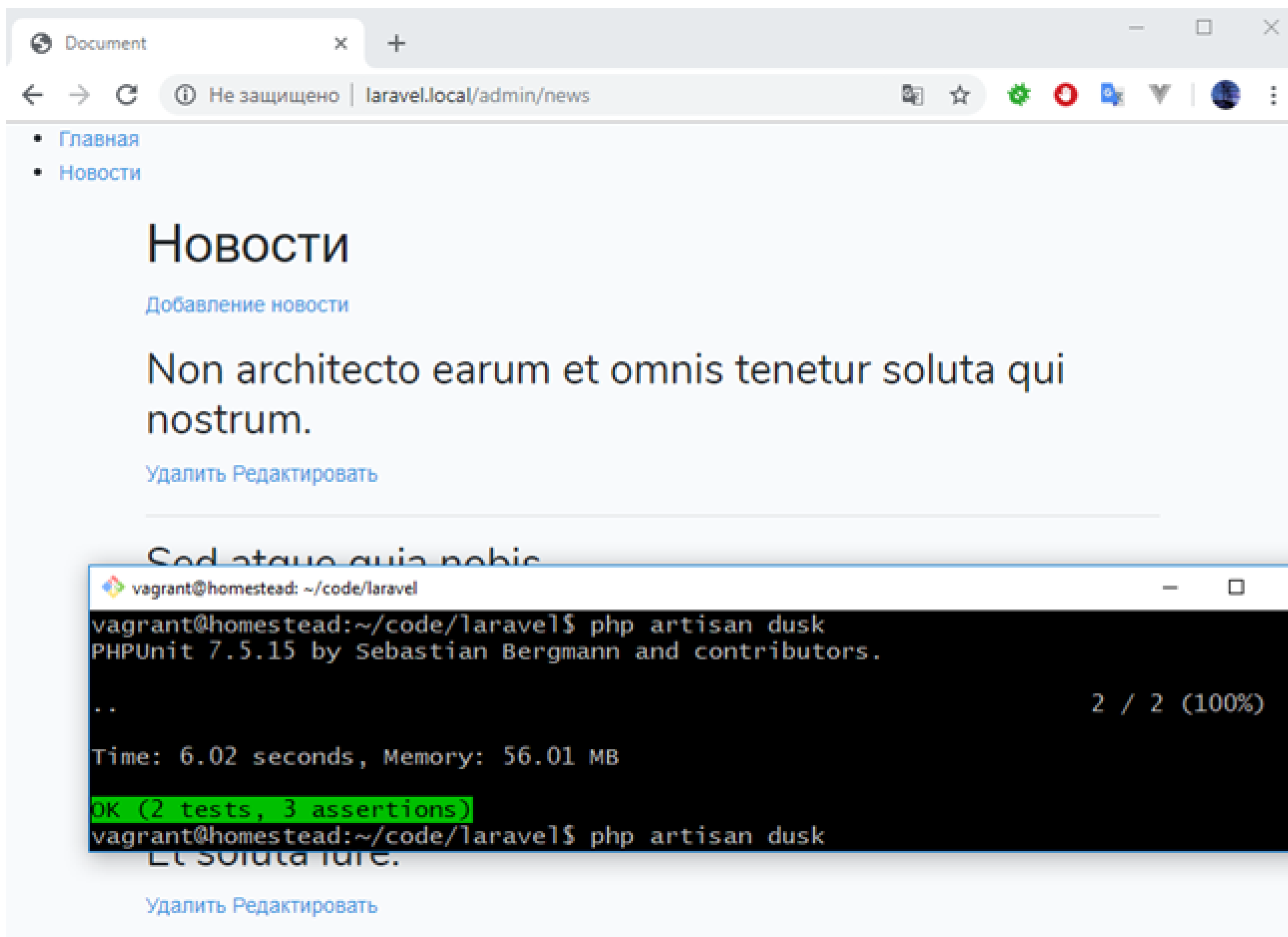
После этого создадим файл **sqlite.testing.database**, в который и будет выполняться запись миграций перед тестированием, а затем запись новой новости во время теста.



Чтобы перед тестами в новую базу данных выполнялись миграции, в класс **AddNewsTest.php** добавим трейт **Illuminate\Foundation\Testing\RefreshDatabase**.



Чтобы при запуске приложения использовался файл **env.dusk**, тест запускаем из консоли: **php artisan dusk**



Обратим внимание, что новая запись в основную базу добавлена не была.

Практическое задание

1. Добавить валидирование данных, которые получены из форм, созданных на предыдущих уроках.
2. Используя знания, полученные на уроке, реализуйте вывод сообщений об ошибках валидации полей форм (из задания 1).
3. Добавьте тесты — минимум по две проверки на каждую форму.

Дополнительные материалы

1. https://laravel.com/docs/5.8/validation.
2. https://laravel.ru/.
3. https://laravel.com/docs/5.8/localization.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://laravel.com/docs/5.8/homestead.
2. http://laravel.su/.