Laravel

Урок 6. Работа с базой данных. Eloquent ORM

Работа с ORM — функциональным и удобным механизмом взаимодействия с БД для приложений и агрегатора.

Оглавление

Теория

- Eloquent.
- Laravel Debugbar

Практика

- Создание моделей.
- Использование конструктора запросов в моделях.
- Постраничный вывод данных.
- Немного о коллекциях в Laravel.
- Операции CRUD с моделями.
- Связи между таблицами.

Практическое задание

Дополнительные материалы

Используемая литература

© geekbrains.ru

Теория

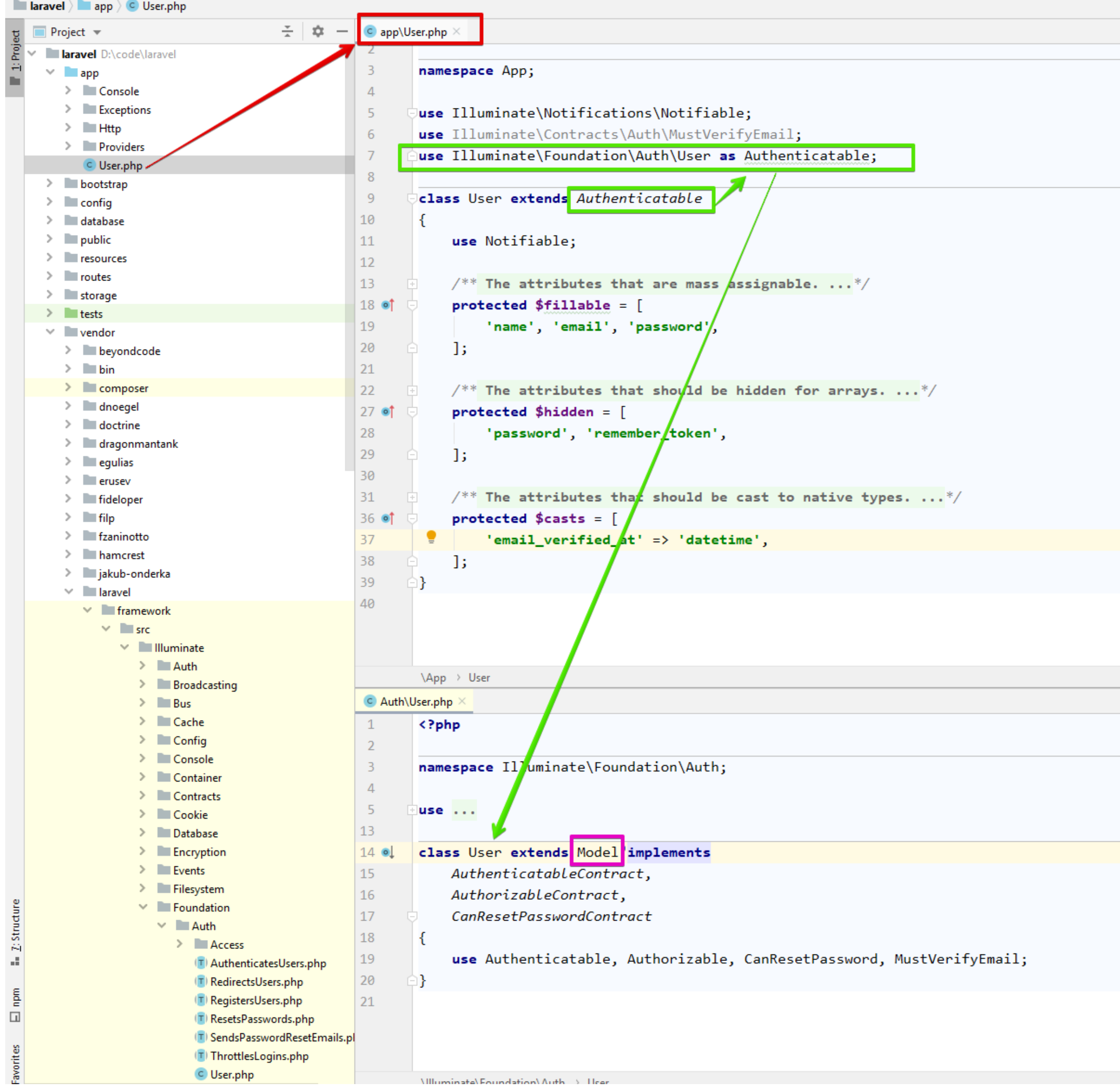
Eloquent

Для работы с запросами к базе данных в Laravel удобно использовать систему объектно-реляционного отображения (ORM) — Eloquent. Это реализация паттерна ActiveRecord (AR).

Основная идея AR: для каждой таблицы в базе данных существует отдельный класс. Он является представлением данных этой таблицы, а отдельный объект этого класса — отражением только одной записи из этой таблицы. При этом каждый такой класс позволяет выполнять CRUD. Группу таких классов в Laravel называют моделью.

По умолчанию модели находятся в папке **app** и носят имя таблицы, которую они представляют. Название таблицы в базе данных, как правило, дается во множественном числе, а имя соответствующей ей модели — в единственном. Если имя таблицы состоит из нескольких слов, то в базе они должны быть записаны через _, а в имени класса каждое новое слово должно указываться с большой буквы. Но эти правила — только рекомендации. Подробнее: https://laravel.ru/docs/v5/eloquent.

Каждая модель Laravel должна быть унаследована от класса **Model**. В проекте уже создана одна модель — класс **User**.



Класс модели может содержать как специфические методы для работы с данными, так и свойства, которые позволяют переопределять параметры работы с таблицами в базе:

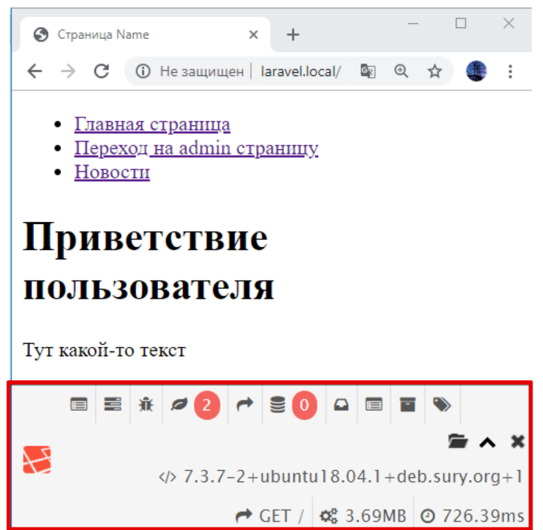
- `protected $table = 'my_flights';` — свойство, переопределяющее название таблицы для модели;
- `protected $primaryKey = 'flight_id';` — свойство для переопределения первичного ключа;
- `public $timestamps = false;` — задавая данному свойству такое значение, указываем на то, что в таблице нет полей, отвечающих за сохранение информации об изменении и создания записи;
- `protected $dateFormat = 'U';` — это свойство используется для указания формата сохранения даты в таблице.

Laravel Debugbar

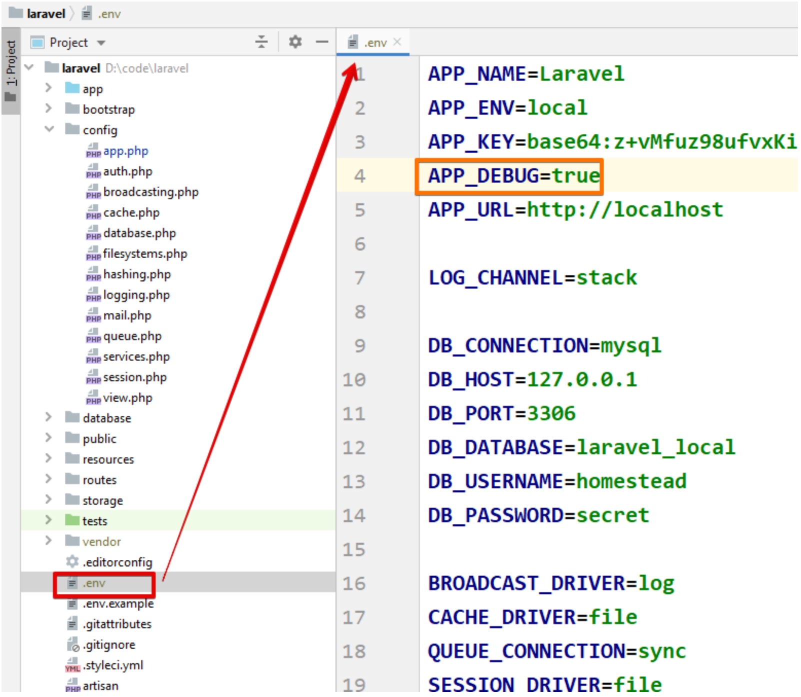
Фреймворк — это набор готовых классов и алгоритмов. Чтобы понимать, что в нем происходит при различных действиях программиста и пользователя, можно воспользоваться замечательным инструментом — Laravel Debugbar. Для установки этого пакета выполним следующую команду **composer require barryvdh/laravel-debugbar –dev**

Когда указанный пакет скачается, добавим в файл **app.php** в папке **config** строку **Barryvdh\Debugbar\ServiceProvider::class,** в массив **providers** и строку **'Debugbar' => Barryvdh\Debugbar\Facade::class,** в массив **aliases**.

Если все сделано верно — внизу страницы появится отображение данного компонента.



Отключить его можно, изменив переменную окружения **APP_DEBUG** на **false**:

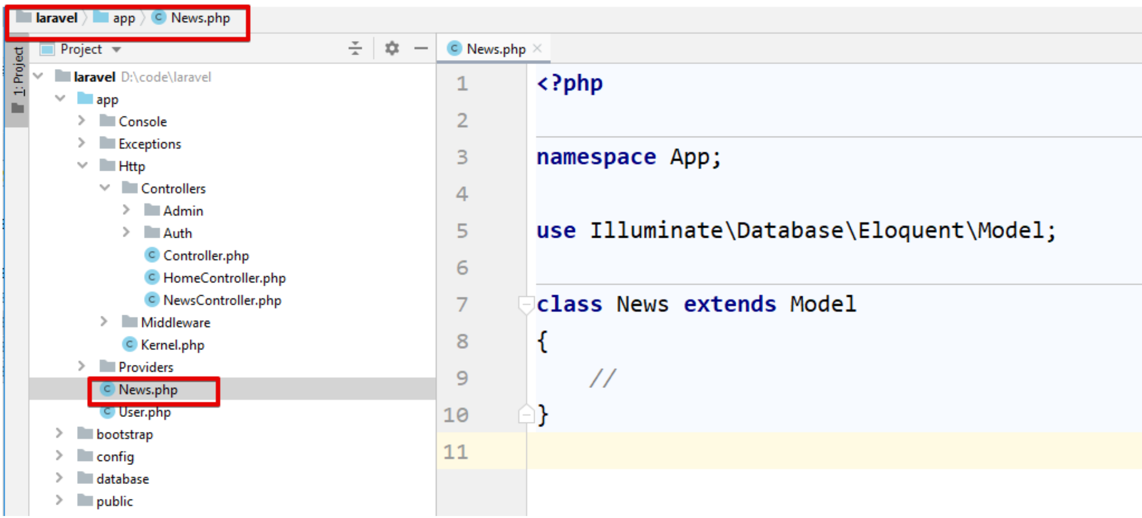


Практика

Создание моделей

В нашем приложении уже создана таблица **news**. Добавим для этой таблицы модель. Для этого воспользуемся следующей терминальной командой: **php artisan make:model News**.

Выполнив ее, перейдем в папку **app/Http** и — в ней появился новый файл **News.php**.

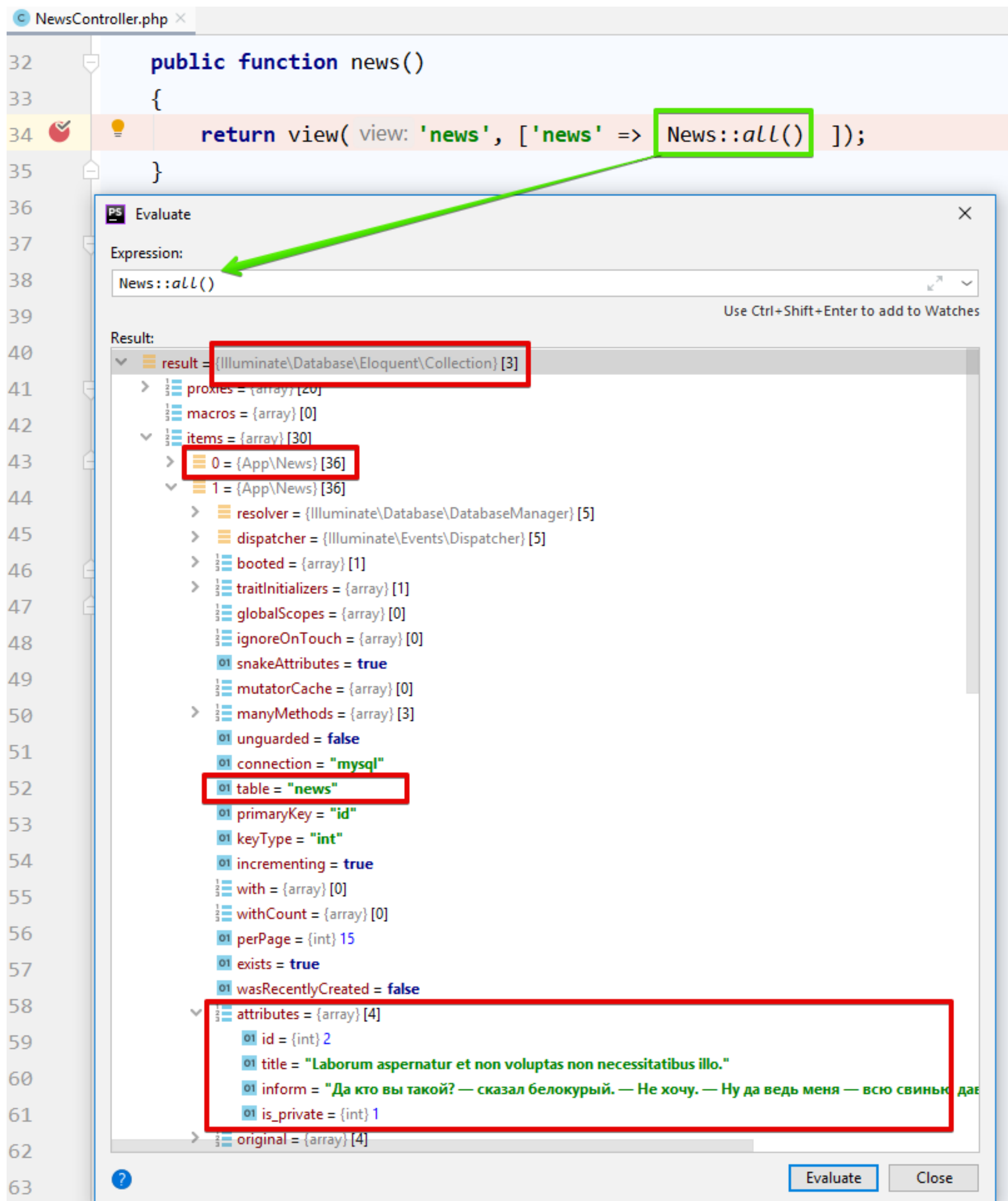


Данный файл содержит класс **News**, который наследуется от класса **Illuminate\Database\Eloquent\Model**.

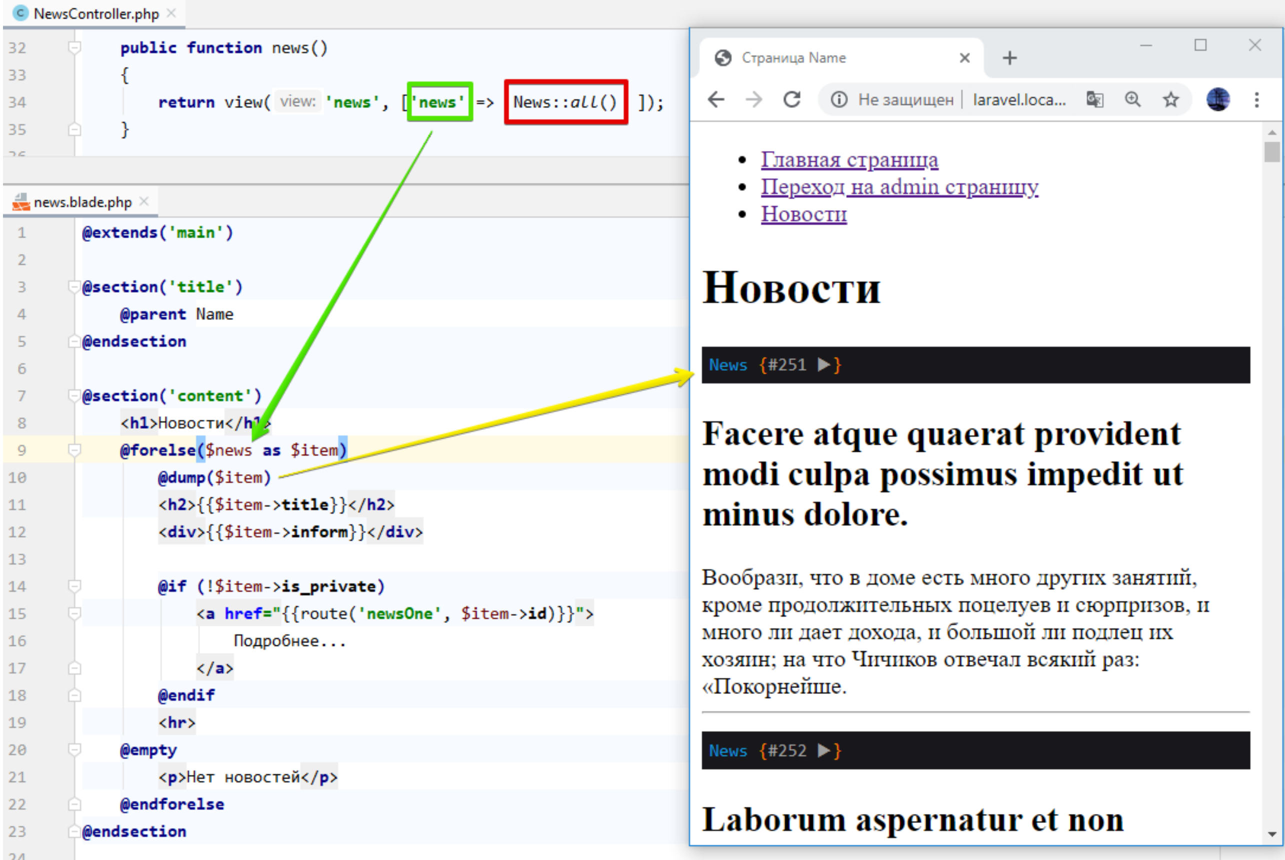
Посмотрите на методы, которые содержатся в этом классе.

Теперь внесем небольшое изменение в **NewsController** — в метод, отвечающий за вывод всех новостей. На скриншоте ниже показано данное изменение и результат его выполнения.

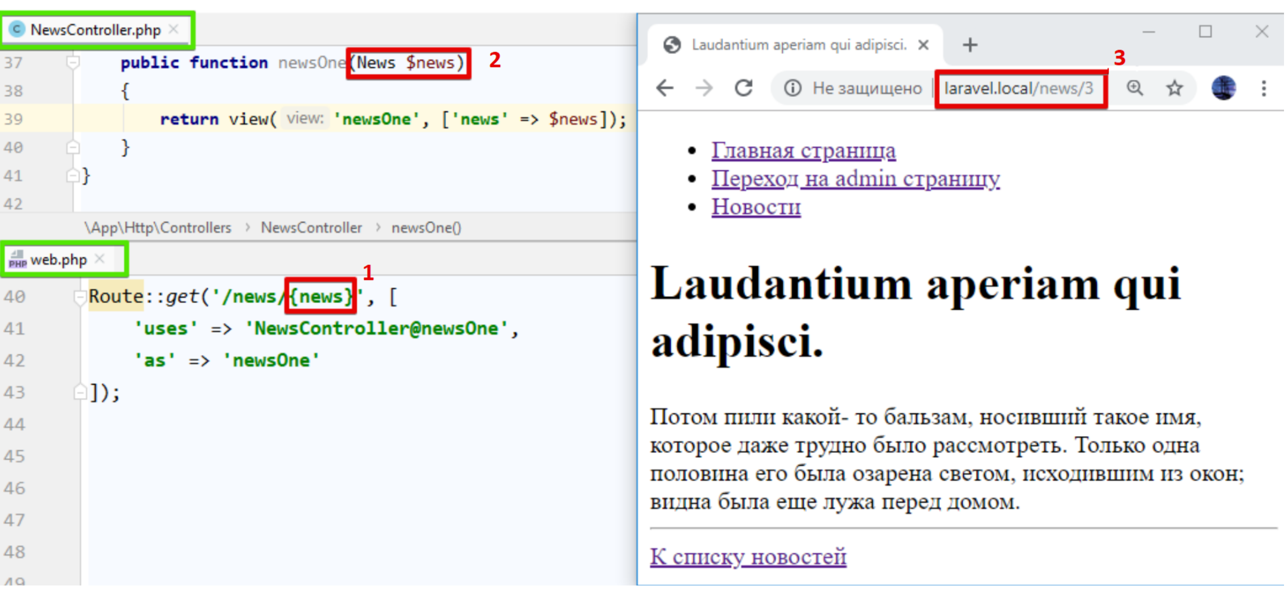
Видим, что у класса **News** (модели для таблицы **news**) есть статичный метод **all**. В результате выполнения данного метода мы получаем объект класса **Illuminate\Database\Eloquent\Collection**. В его свойстве **items** содержится массив из объектов класса **News**. Внутри каждого объекта класса **News** есть информация отдельной записи из таблицы **news**.



Класс **Illuminate\Database\Eloquent\Collection** реализует встроенный в PHP интерфейс, который обеспечивает доступ к объектам в виде массивов (**ArrayAccess**). Поэтому объект данного класса можем перебрать при помощи обычного **foreach** (**forelse**).



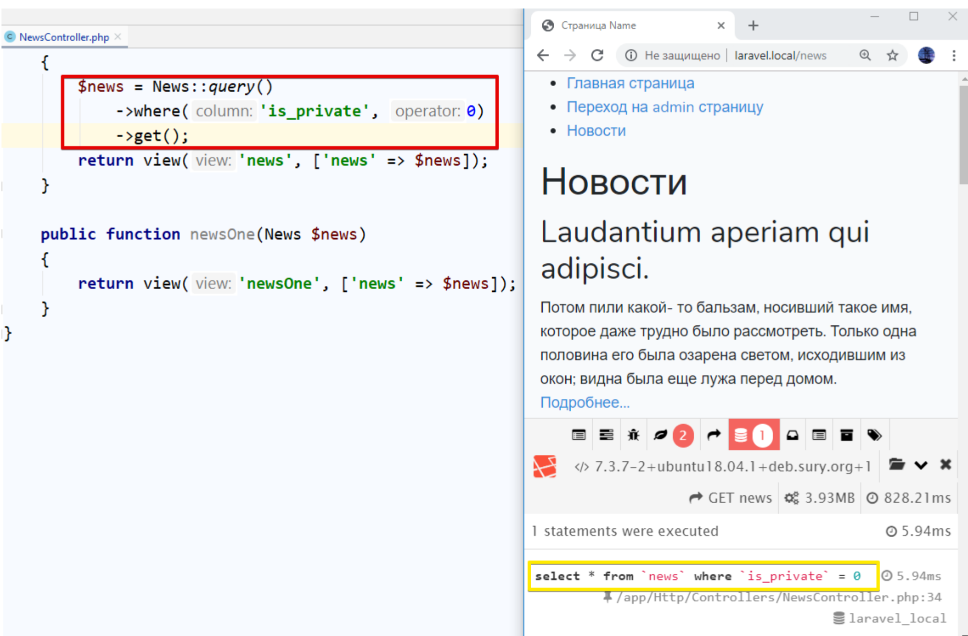
А теперь посмотрим на магию моделей в Laravel.



Как это работает?! Когда в роуте имя параметра — это название модели, а в методе контроллера указывается, что ожидается объект данной модели, то Laravel производит поиск в базе данных. Запись ищется по значению первичного ключа — равного тому, что передан в адресной строке в качестве этого параметра. В случае, представленном выше, производится поиск в таблице **news** записи с **id = 3**. Если запись не будет найдена — появится ошибка 404 и соответствующая ей страница. Если запись найдена — объект указанной модели будет заполнен данными и передан в метод в качестве параметра.

Использование конструктора запросов в моделях

Модели не ограничивают нас в использовании конструктора запросов. Если надо указать, что в выборке следует вывести только данные неприватных новостей, то следует обратиться к статическому методу **query**. Он создаст конструктор запросов для указанной модели, после чего можно использовать методы конструктора.



Постраничный вывод данных

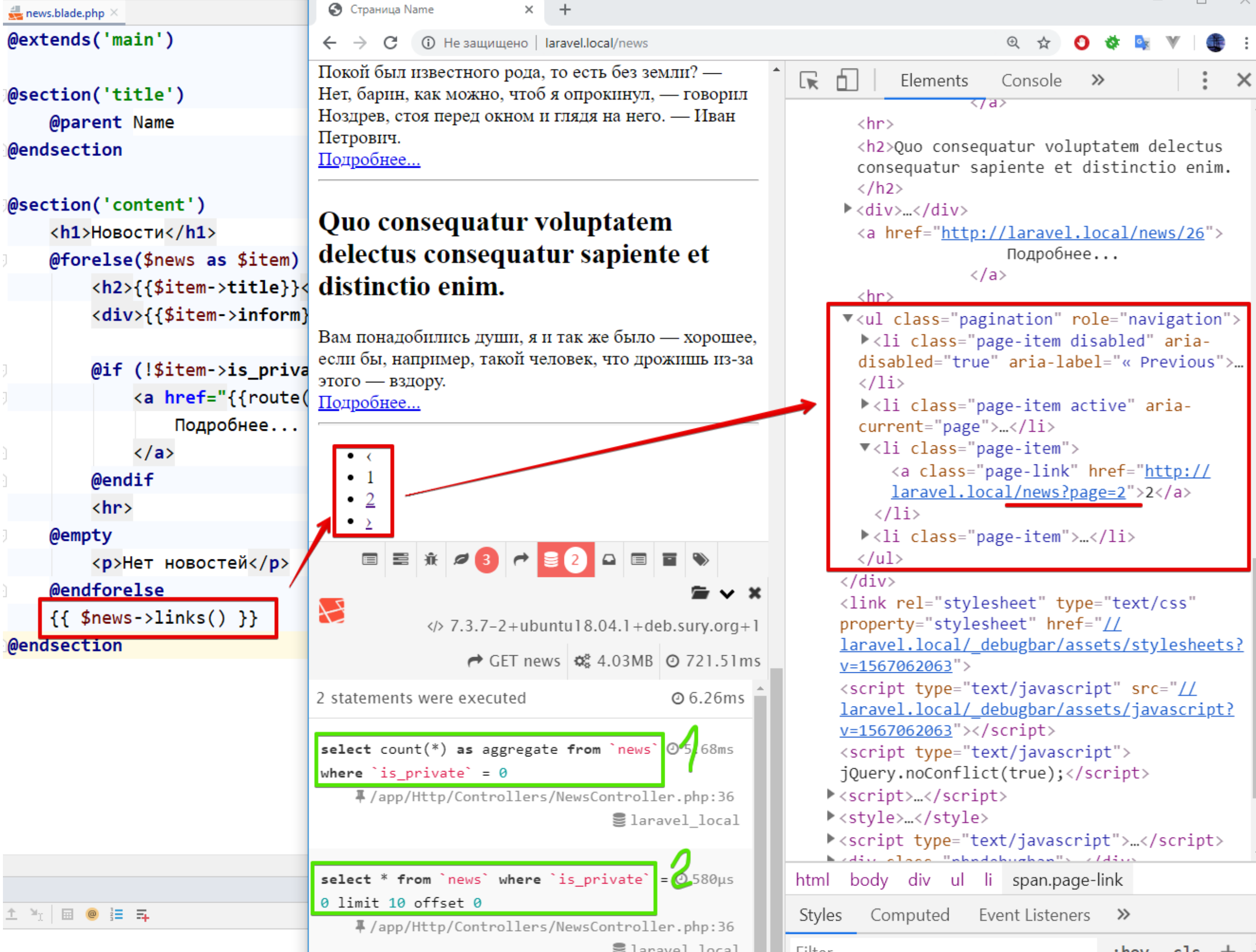
Для постраничного вывода (пагинации) в Laravel встроен удобный механизм, который в несколько строк решает эту задачу. Запуск алгоритма пагинации происходит в тот момент, когда в конструкторе запроса используется метод **paginate**. Laravel сначала делает запрос на определение количества записей в таблице, после чего обращается к адресной строке, ожидая найти в ней get-параметр **page**. По отсутствию параметра или указанному в нем значению Laravel определяет, какая страница должна быть выдана. Метод **paginate** возвращает объект класса **Illuminate\Pagination\LengthAwarePaginator**:

```
public function news()
{
    $news = News::query()
        ->where( column: 'is_private',  operator: 0)
        ->paginate( perPage: 10);
    return view( view: 'news', ['news' => $news]);
}

public function newsOne(News $news)
{
    return view( view: 'newsOne', ['news' => $news]);
}
}
```

```
Evaluate
Expression:
News::query()->where('is_private', 0)->paginate(10)
Use Ctrl+Shift+Enter to add to Watches
Result:
result = {Illuminate\Pagination\LengthAwarePaginator} [16]
  currentPathResolver = {Closure} [1]
  currentPageResolver = {Closure} [2]
  viewFactoryResolver = {Closure} [1]
  defaultView = "pagination::bootstrap-4"
  defaultSimpleView = "pagination::simple-bootstrap-4"
  total = {int} 13
  lastPage = {int} 2
  items = {Illuminate\Database\Eloquent\Collection} [3]
  perPage = {int} 10
  currentPage = {int} 1
  path = "http://laravel.local/news"
  query = {array} [0]
  fragment = null
  pageName = "page"
  onEachSide = {int} 3
  options = {array} [2]
```

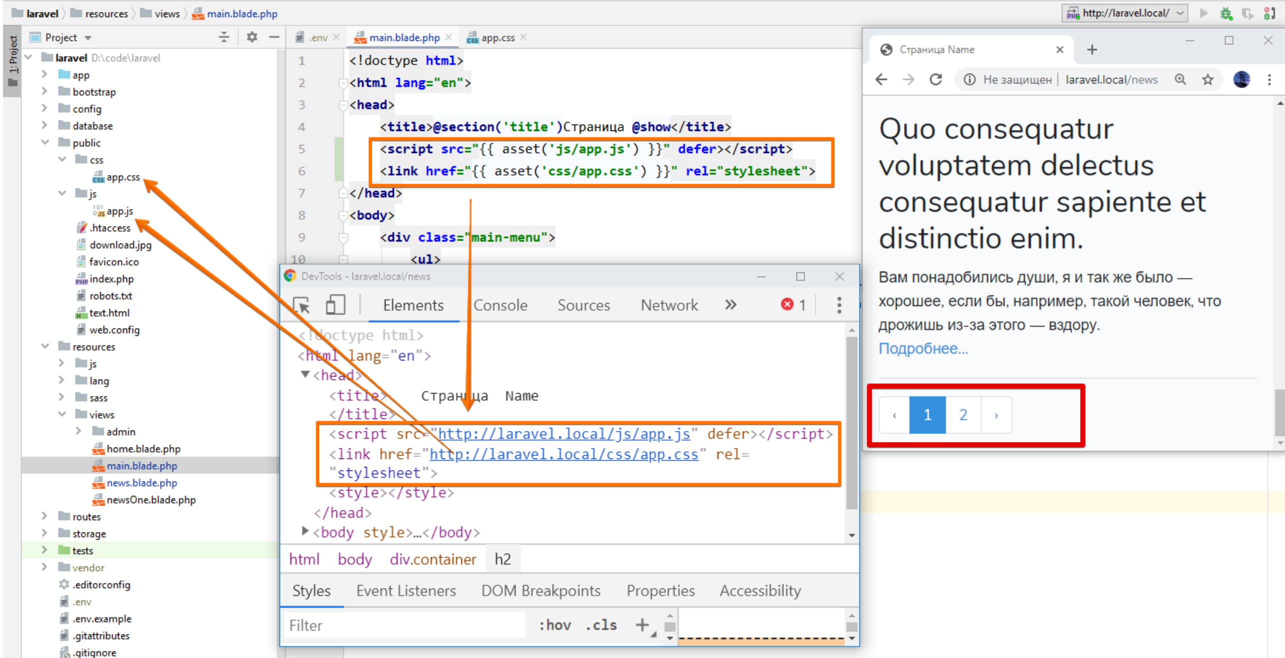
Класс **Illuminate\Pagination\LengthAwarePaginator** тоже реализует интерфейс **ArrayAccess**, что позволяет перебирать его свойства, как элементы массива. Поэтому вывод в шаблоне менять не придется.

Обратим внимание на метод **links**. Добавим его вызов в шаблон **news.blade.php**. В результате заметим, что на странице появилась дополнительная html-разметка, которая содержит ссылки, передающие значение для get-параметра **page**.



В Debugbar, открыв вкладку Queries, сможем увидеть запросы, которые были выполнены при запросе данной страницы (отмечены на рисунке цифрами 1 и 2).

Добавим немного красоты: для этого в **main.blade.php** подключим стили и **js**:



Немного о коллекциях в Laravel

При выполнении некоторых методов конструктора запросов возвращается объект класса **\Illuminate\Database\Eloquent\Collection**. Этот класс — полезный инструмент, который позволяет оперировать данными внутри него. Коллекцию можно создать и самостоятельно: для этого следует воспользоваться хелпером **collect** и передать в него необходимый набор элементов коллекции.

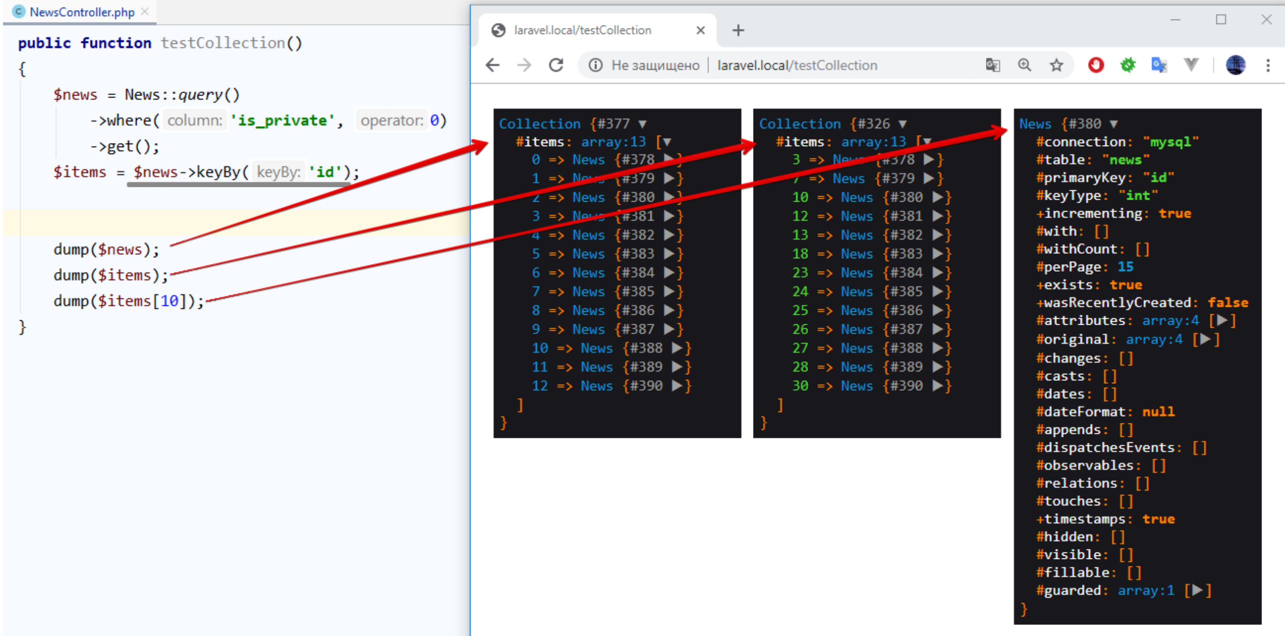
```
public function testCollection()
{

    $collection = collect([1,2,3]);
    dump($collection);

}
```

```
Collection {#326 ▼
  #items: array:3 [▼
    0 => 1
    1 => 2
    2 => 3
  ]
}
```

Рассмотрим несколько методов данного класса:

- **count**, **min**, **max**, **last**, **first**, **sort** — методы, о назначении которых легко догадаться по их именам;

- **keyBy** — позволяет получить коллекцию, ключи которой будут соответствовать полю модели, указанному в качестве параметра при вызове этого метода;



- **diff** — позволяет получить коллекцию, которая будет разницей между исходной и переданной в качестве параметра.

Полный набор методов можно посмотреть в документации: https://laravel.com/docs/5.8/collections.

Операции CRUD с моделями

Для начала сделаем нашу админку приятнее. Для этого внесем несколько корректировок в шаблоны — прикрепим стили и добавим классы.

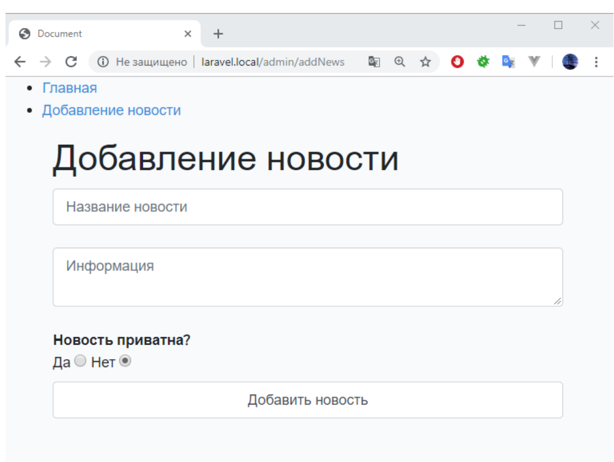
main.blade.php

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>@section('title')Document @show</title>
    <script src="{{ asset('js/app.js') }}" defer></script>
    <link href="{{ asset('css/app.css') }}" rel="stylesheet">
</head>
<body>
    <ul>
        <li><a href="{{route('home')}}">Главная</a></li>
        <li><a href="{{route('admin.addNews')}}">Добавление новости</a></li>
    </ul>

    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

addNews.blade.php

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

@section('content')
    <h1>Добавление новости</h1>
    <form action="{{route('admin.addNews')}}" method="post">
        @csrf
        <input class="form-control" name="title" placeholder="Название новости" value="{{old('title')}}"> <br>
        <textarea class="form-control" name="inform" placeholder="Информация">{{ old('inform') }}</textarea> <br>
        <b>Новость приватна?</b> <br>
        <label>Да
            <input name="isPrivate" type="radio" value="1" @if (old('isPrivate') == 1) checked @endif>
        </label>
        <label>Нет
            <input name="isPrivate" type="radio" value="0" @if (old('isPrivate') == 0) checked @endif>
        </label> <br>
        <button class="form-control" type="submit">Добавить новость</button>
    </form>
@endsection
```

В итоге получим что-то такое:



Добавим несколько маршрутов, которые будут отвечать за редактирование новостей, их вывод и удаление:

```
Route::group(
    [
        "prefix" => "admin",
        "namespace" => "Admin",
        "as" => "admin."
    ],
    function () {
        Route::get('/', [
            'uses' => 'IndexController@index',
            'as' => 'index'
        ]);
        Route::get('/news', [
            'uses' => 'NewsController@all',
            'as' => 'news'
        ]);
        Route::match(['post', 'get'], '/addNews', [
            'uses' => 'NewsController@add',
            'as' => 'addNews'
        ]);
        Route::match(['post', 'get'], '/updateNews/{news}', [
            'uses' => 'NewsController@update',
            'as' => 'updateNews'
        ]);
        Route::match(['post', 'get'], '/deleteNews/{news}', [
            'uses' => 'NewsController@delete',
            'as' => 'deleteNews'
        ]);
    }
);
```

Создадим новый шаблон для вывода всех новостей в админке. Для каждой новости добавим ссылки на удаление и редактирование:

Теперь добавим соответствующие методы в **NewsController** (пока это в основном заглушки):

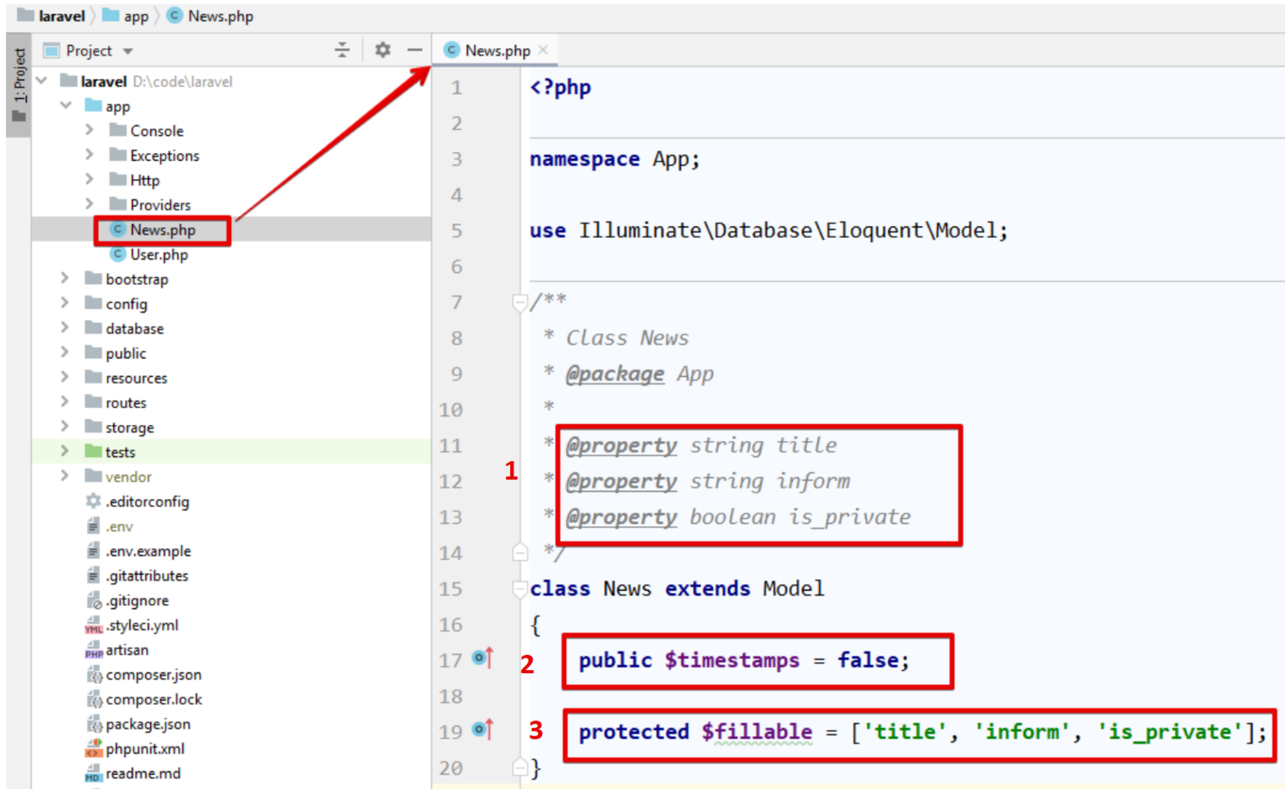
```php
use App\News;

class NewsController extends Controller
{
    public function all()
    {
        $news = News::query()
            ->orderBy( column: 'id',  direction: 'desc')
            ->paginate( perPage: 5);
        return view( view: 'admin.allNews', ['news' => $news]);
    }

    public function add(Request $request)
    {
        if ($request->isMethod( method: 'post')) {
            $request->flash();
            return redirect()->route( route: 'admin.addNews');
        }
        return view( view: 'admin.addNews');
    }

    public function update(Request $request, News $news)
    {
        dump($news);
    }

    public function delete(News $news)
    {
        dump($news);
    }
}
```

Внесем правки в шаблон для добавления новостей:

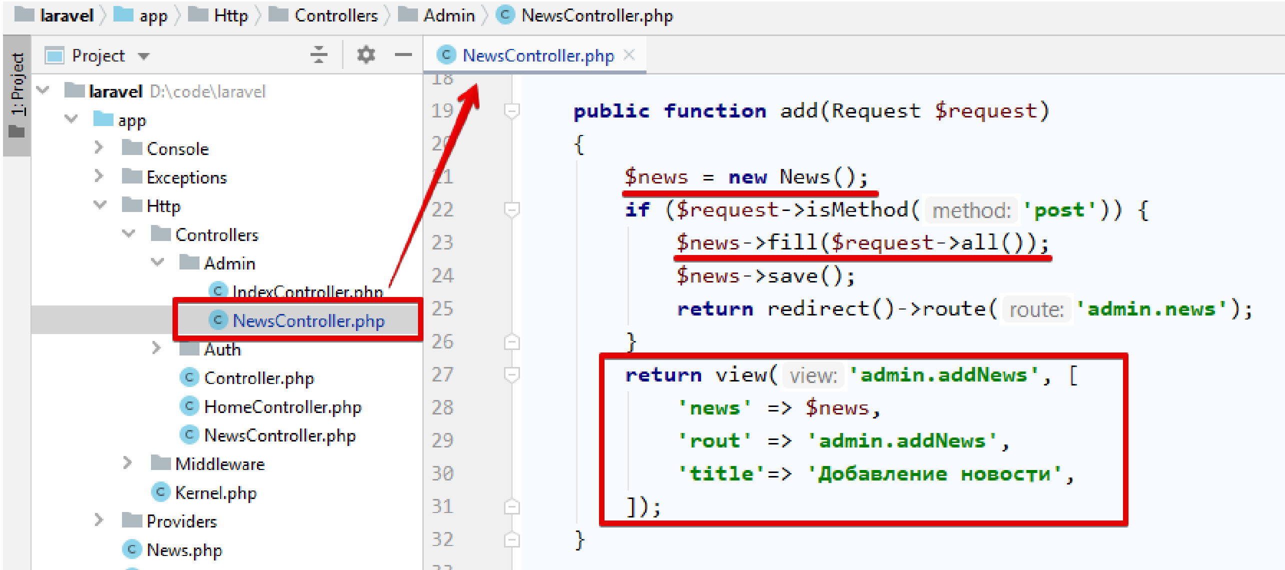
```php
@extends('admin.main')

@section('content')
    <h1>{{$title}}</h1>
    <form action="{{route($rout, $news->id)}}" method="post">
        @csrf
        <input class="form-control" name="title" placeholder="Название новости" value="{{$news->title}}"> <br>
        <textarea class="form-control" name="inform" placeholder="Информация">{{ $news->inform }}</textarea> <br>
        <b>Новость приватна?</b> <br>
        <label>Да
            <input name="is_private" type="radio" value="1" @if ($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
```

Внесем изменения в модель **News**:

- добавим аннотацию для получения подсказок от **ide**;
- укажем для Laravel, что наша таблица не содержит информации о дате создания и изменения данных;
- создадим свойство, которое будет возвращать названия полей нашей таблицы для вставки данных.

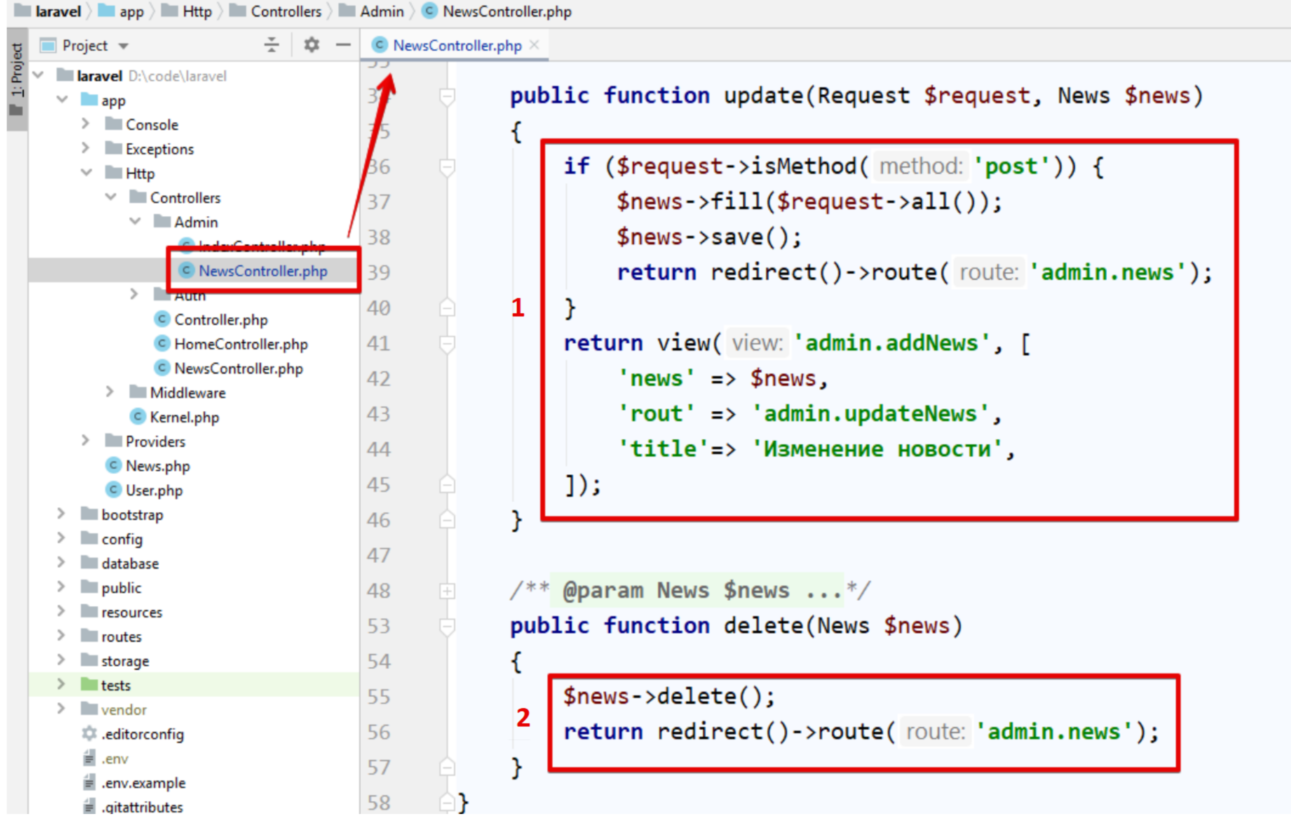


Внесем правки в **NewsController**. Добавим создание нового экземпляра класса **News** и вызов метода на его заполнение (**fill**). Этот метод в качестве параметра принимает массив данных и заполняет поля модели, указанные в свойстве **$fillable**. Также добавим необходимые данные для шаблона.



На этом алгоритм для добавления новых новостей завершен.

Реализуем метод для редактирования и удаления новости:

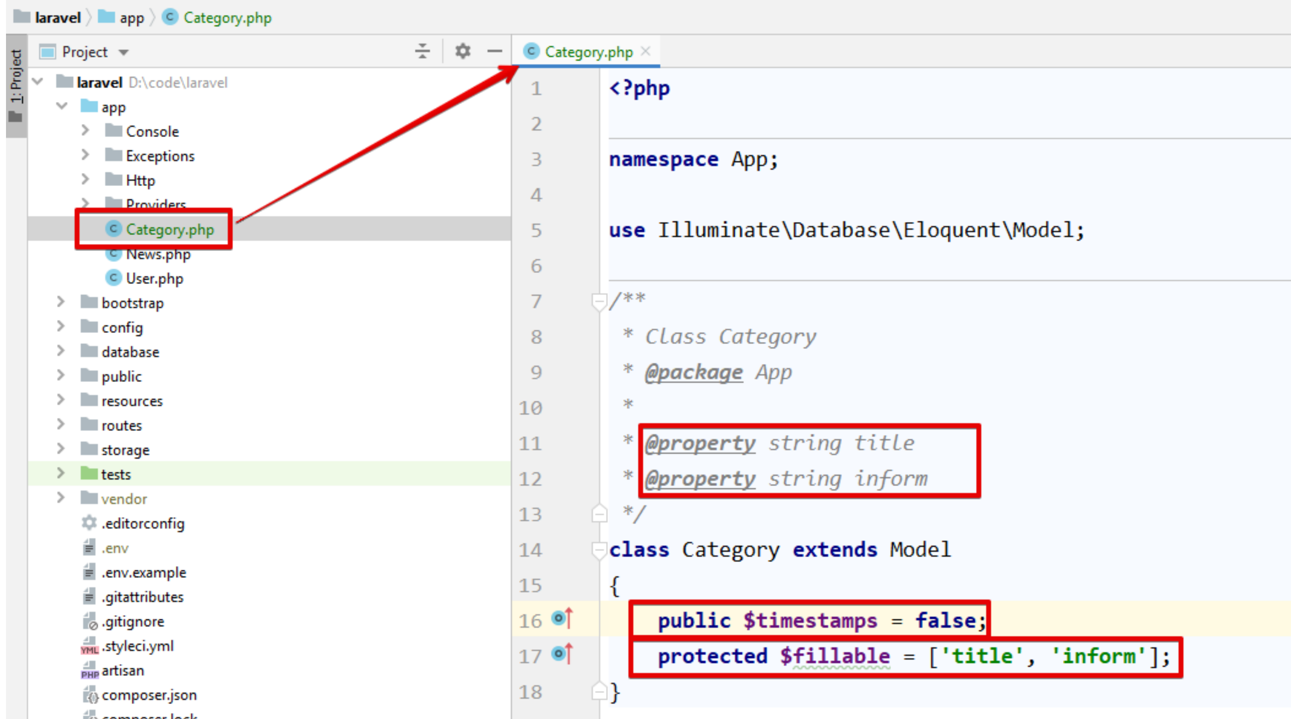


Связи между таблицами

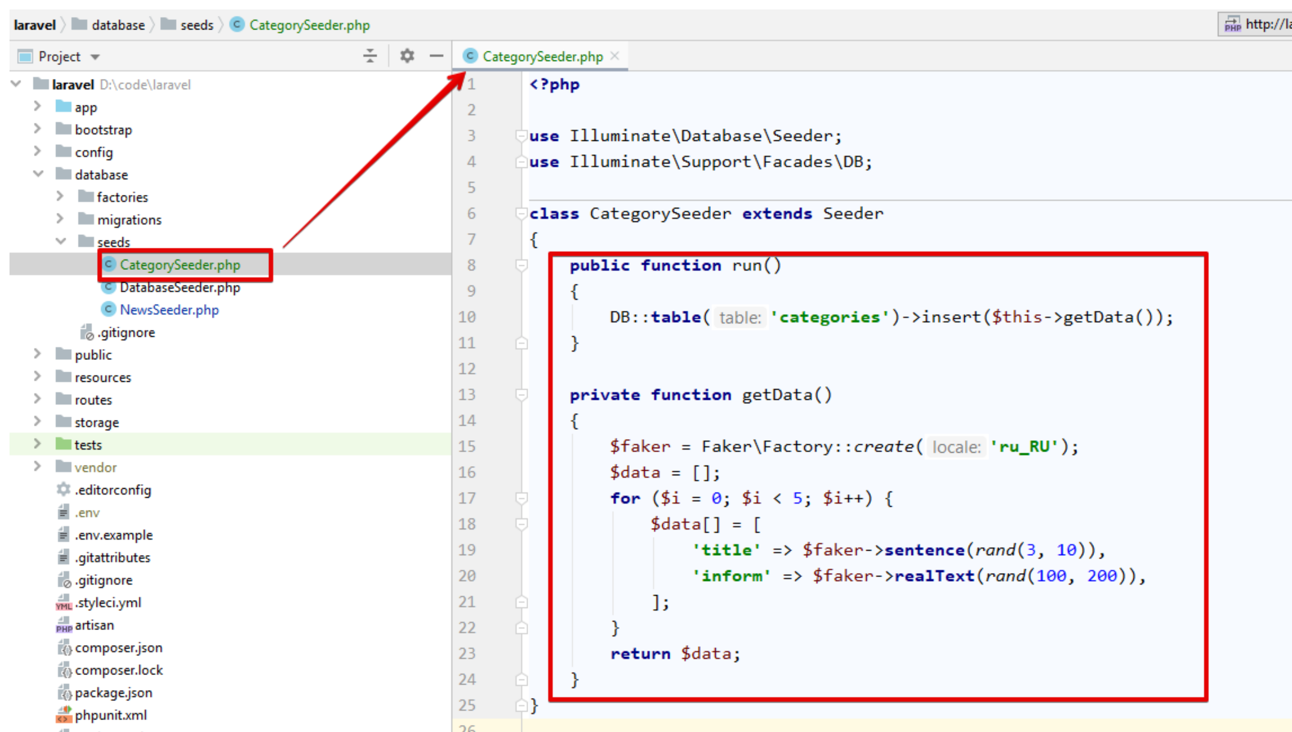
Предположим, нам необходимо выводить новости не целым списком, а категориями. Для этого понадобится создать отдельную таблицу категорий и для каждой записи в таблице с новостями указать, к какой категории она будет относиться.

Создадим новую миграцию, которая добавит таблицу категорий, и реализуем модель для нее. Для этого выполним команду **php artisan make:model Category —m**. Ключ **–m** создаст шаблон для миграции.

В созданной модели добавим аннотацию. Укажем, что в ней не будет использоваться **timestamps**, а также отметим, какие поля будут доступны для заполнения свойством **fillable**.



Прежде чем создать миграцию, добавим агрегатор для тестовых данных. Для этого создадим **seeder** командой **php artisan make:seeder CategorySeeder**. В созданный класс добавим следующую инструкцию:



В созданной миграции добавим следующий код:

```php
    {
        public function up()
        {
            Schema::create( table: 'categories', function (Blueprint $table) {
                $table->bigIncrements( column: 'id');
                $table->string( column: 'title')->comment( comment: 'Название категории');
                $table->text( column: 'inform')->comment( comment: 'Описание категории');
            });
            (new CategorySeeder)->run();
        }

        public function down()
        {
            Schema::dropIfExists( table: 'categories');
        }
    }
```

Обратите внимание на строку 16: тут произойдет выполнение заполнения тестовыми данными из ранее созданного посева.

Теперь можно выполнить добавление поля категории в таблицу **news**. Для этого создадим еще одну миграцию: **php artisan make:migration alter_table_news_add_category**

```php
<?php

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

class AlterTableNewsAddCategory extends Migration
{
    public function up()
    {
        Schema::table( table: 'news', function (Blueprint $table) {
            $table->unsignedBigInteger( column: 'category_id')->default( value: 1);
            $table->foreign( columns: 'category_id')
                ->references( columns: 'id')
                ->on( table: 'categories');
        });
    }

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

Выполним миграции командой **php artisan migrate**

В итоге получаем две связи между таблицами:

- один ко многим — у одной категории множество новостей;
- многие к одному — для множества новостей используется одна категория.

Далее добавим информацию о созданных связях в классы-модели **News** и **Category**.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

/**
 * Class News
 * @package App
 *
 * @property string title
 * @property string inform
 * @property boolean is_private
 */
class News extends Model
{
    public $timestamps = false;

    protected $fillable = ['title', 'inform', 'is_private', 'category_id'];

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

В примере выше добавлен метод **category**, который будет возвращать объект класса **Category**, соответствующий используемой модели. В свойство **fillable** добавлено дополнительное значение, соответствующее названию поля для связи с таблицей категорий.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

/**
 * Class Category
 * @package App
 *
 * @property string title
 * @property string inform
 */
class Category extends Model
{
    public $timestamps = false;
    protected $fillable = ['title', 'inform'];

    public function news()
    {
        return $this->hasMany( related: News::class,  foreignKey: 'category_id');
    }
}
```

В класс **Category** добавлен метод для получения связанных моделей **News**. Данный метод возвращает объект класса **Illuminate\Database\Eloquent\Relations\HasMany**, который позволяет при помощи конструктора запроса сделать выборку необходимых записей из таблицы **News**. Пример использования объекта данного класса:

```
Category::query()->find(1)->news()->where('id', '>', 12)->get();
Category::query()->find(1)->news()->get();
```

Теперь внесем небольшие правки в контроллер и в шаблон, чтобы указывать категории в новости при создании и редактировании. Задача — добавить в форму select-элемент, в котором будет происходить выборка нужной категории. Значит, нам нужно получить все категории.

```
use App\Category;

class NewsController extends Controller
{
    public function all(){...}

    public function add(Request $request)
    {
        $news = new News();
        if ($request->isMethod( method: 'post')) {
            $news->fill($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(),
        ]);
    }

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

В шаблоне с помощью цикла выведем все возможные категории. Если id категории будет совпадать с категорией в новости, то укажем, что она должна быть выбрана.

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

@section('content')
    <h1>{{$title}}</h1>
    <form action="{{route($rout, $news->id)}}" method="post">
        @csrf
        <input class="form-control" name="title" placeholder="Название новости" value="{{$news->title}}"> <br>
        <textarea class="form-control" name="inform" placeholder="Информация">{{ $news->inform }}</textarea> <br>

        <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
```

Практическое задание

1. Добавить модели для используемых в проекте таблиц, созданных на предыдущих занятиях. Если нужно изменить структуру таблицы — создать миграции.
2. При помощи миграций создать таблицы для сохранения данных из форм, созданных на предыдущих уроках. Реализовать возможность сохранения информации из этих форм в соответствующие таблицы.
3. Реализовать возможность добавления, редактирования и удаления для таблиц, содержащих:
    a. Новости.
    b. Категории.
    c. Данные обратной связи.
    d. Заказы на получение выгрузки данных.
4. * Дополнительное задание: реализовать удаление данных из перечисленных таблиц, используя в маршруте для удаления http-метод DELETE.

Дополнительные материалы

- https://laravel.com/docs/5.8/eloquent.
- https://laravel.com/docs/5.8/eloquent-relationships.
- https://github.com/barryvdh/laravel-debugbar.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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