**ЗАО "БелХард Групп"**

**Центр обучающих технологий**

**Михалькевич Александр Викторович**

**Laravel**

****

**Фрэймворк для web-ремесленников**

**2-ое издание**

**Минск 2015**

УДК 004.415.53

ББК 32.973.2-018.2я7

Рецензент:

Об авторе:

Михалькевич Александр Викторович, программист, преподаватель учебных курсов по PHP.

Михалькевич Александр Викторович

PHP для продвинутых Laravel

Михалькевич Александр Викторович, Закрытое акционерное общество «БелХард Групп» - Мн., 2015. – 269с.

ISBN

* Построение web-приложения на Laravel

Пособие рекомендовано к использованию слушателям курсов ЦОТ ЗАО "БелХард Групп".

УДК 004.415.53

ББК 32.973.2-018.2я7

© Михалькевич Алексндр Викторович., 2015

© Закрытое акционерное общество

ISBN «БелХард Групп», 2015

**Оглавление**

1. Инструментарий------------------------------------------------------------------------3
2. Laravel Homestead ----------------3
3. Установка laravel-----------------------------------------------------3
4. Удаление public из url проекта--------------------------------------7
5. Настройка----------------------------------------------------8
6. Маршрутизация------------------------------------------------12
7. Middleware
8. Пространства имен: основы--------------------------11
9. Контроллеры----------------------------------------------------13
10. Шаблоны----------------------------------------------------------16
11. База данных MySQL, миграции-----------------------------------------17
12. Модели-------------------------------------------------------------22
13. Модуль авторизации------------------------------------------------------29
14. Постраничная навигация-----------------------------------32
15. Модуль обработки изображений----------------------------33
16. Profiler Debugbar--------------------------------------------------34
17. Artisan-----------------------------------------34
18. Service Providers------------------------------------------------35

# 1. Инструментарий

**IDE NetBeans** – Интегрированная среда разработки.

**Nodepad++** .

**OpenServer** – Локальный сервер (Содержит встроенные технологии:PHP, Apache, MySQL и системы управления MySQL, Composer и др.).

**Git** – система контроля версий

**Composer** – локальный менеджер зависимостей.

**Firefox** – Браузер.

**Firebug** – Отладчик кода на стороне клиента

**FireFTP** – FTP.

**2. Laravel Homestead**

Vagrant предоставляет простой и элегантный способ создания и управления виртуальными машинами. Laravel Homestead является официальным "образом" (box) для Vagrant'а, и предоставляет замечательную среду разработки, не требуя устанавливать PHP, веб-сервер и какое бы то ни было дополнительное серверное ПО на вашей локальной машине. Больше не стоит беспокоиться о захламлении вашей операционной системы! Боксы Vagrant'а являются полностью одноразовыми. Если что-то пойдет не так, вы сможете уничтожить и пересоздать бокс за считанные минуты!

Homestead работает под любыми версиями Windows, Mac и Linux, и включает веб-сервер Nginx, PHP 5.6, MySQL, Postgres, Redis, Memcached и другие вкусности, которые могут потребоваться вам для разработки потрясающих Laravel-приложений.

Текущая версия Homestead создана и протестирована для использования под Vagrant 1.6.

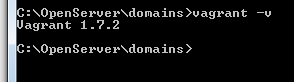
* Что внутри
* Ubuntu 14.04
* PHP 5.6
* Nginx
* MySQL
* Postgres
* Node (включая Bower, Grunt и Gulp)
* Redis
* Memcached
* Beanstalkd
* Laravel Envoy
* Fabric + HipChat Extension

### Установка VirtualBox и Vagrant

Перед запуском среды Homestead, вы должны установить [VirtualBox](https://www.virtualbox.org/wiki/Downloads) и [Vagrant](http://www.vagrantup.com/downloads.html). Оба этих программных продукта имеют легкие в использовании установщики для всех популярных операционных систем.

После установки vagrant, откроем консоль и определим текущую версию vagrant следующей командой:

|  |
| --- |
| **Определение версии vargrant. Листинг 1.** |
| vagrant -v |



Далее, для создания виртуальной машины нам нужна команда *up*, которая создаст виртуальную машину по предворительным настройкам из файла vagrantfile (https://docs.vagrantup.com/v2/vagrantfile/).

|  |
| --- |
| **. Листинг 1.** |
| vagrant up |

Эта команда установит

### Добавление бокса в Vagrant

Как только VirtualBox и Vagrant будут установлены, вам следует добавить бокс laravel/homestead в Vagrant, используя следующую команду в командной строке. Процесс скачки бокса займет какое-то время, в зависимости от скорости вашего интернет-соединения:

|  |
| --- |
| **Добавление бокса vagrant. Листинг 1.** |
| vagrant box add laravel/homestead |

### Установка Homestead

#### С помощью Composer и PHP

На машине должен быть установлен PHP и [Composer](https://getcomposer.org)

После того как бокс Homestead добавлен в Vagrant, при помощи композера установите глобально инструмент Homestead CLI:

|  |
| --- |
| **Установка Homestead CLI. Листинг 1.** |
| composer global require "laravel/homestead=~2.0" |

После установки инструмента, создайте конфигурационный файл Homestead.yaml (C:\Users\username\.homestead):

|  |
| --- |
| **Создание конфигурационного файла homestead.yaml. Листинг 1.** |
| homestead init |

**С помощью git**

В этом варианте вы можете обойтись без установки PHP на локальную машину, вам понадобится только установленный Git+msysgit.

Склонируйте репозиторий с Homestead CLI в произвольную директорию:

|  |
| --- |
| **Установка Homestead CLI с помощью git. Листинг 1.** |
| git clone https://github.com/laravel/homestead.git Homestead |

Для создания файла Homestead.yaml (C:\Users\username\.homestead) выполним команду:

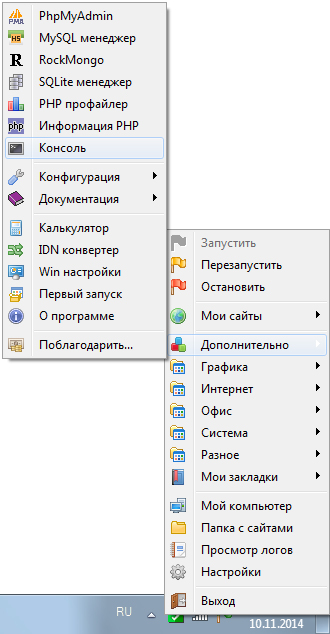
|  |
| --- |
| **. Листинг 1.** |
| bash init.sh |

**3. Установка Laravel**

**Composer**

Laravel использует менеджер зависимостей composer. Если в качестве сервера мы используем OpenServer, то composer поставлен совместно с пакетом программ OpenServer-а. Отдельно его скачивать и устанавливать не нужно.

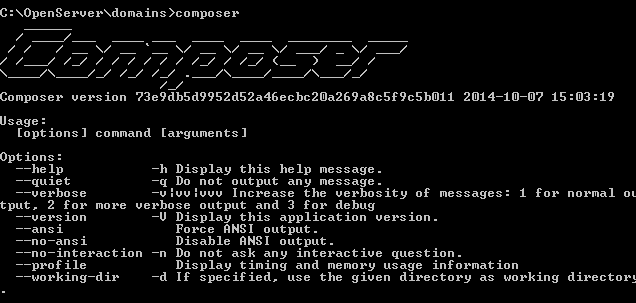
Прежде чем устанавливать Laravel, давайте убедимся, что Composer работает. Для запустим встроенную консоль OpenServer-а.



В консоли наберем команду “Composer”:

|  |
| --- |
| **Команда composer. Листинг 1.2** |
| Composer |

Такой ответ консоли свидетельствует о том, что Composer установлен и готов к работе.



Если с момента последнего запуска composer прошло более 30 дней, то необходимо обновить composer.

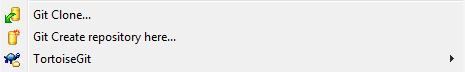
|  |
| --- |
| **Обновление composer. Листинг 1.2** |
| Composer selfupdate |

**GIT**

Также Laravel требует установленной системы контроля версий GIT. GIT скачиваем с официального сайта, и проходим стандартную процедуру установки.

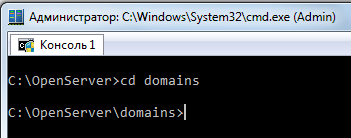
Для удобства работы с GIT, можно установить графическую оболочку TortoiseGit.

После установки TortoiseGit, если правой кнопкой кликнуть по пустому месту папки, то в дополнение к стандартным возможностям добавятся следующие действия:



**Установка Laravel через консль**

Через встроенную консоль OpenServer-а перейдем в корневую дирректорию сервера, либо в ту дирректорию, в которой хотим создать папку с проектом.



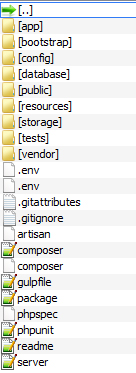
Выполним следующую команду в консоли:

|  |
| --- |
| **Установка laravel. Листинг 1.2** |
| Composer create-project laravel/laravel **-–prefer-dist** |

Либо эту же команду, но с именем папки проекта

|  |
| --- |
| **Установка laravel в определенную папку. Листинг 1.2** |
| Composer create-project laravel/laravel **your-project-name** --prefer-dist |

Laravel5 установит создаст папку laravel, со следующей структурой:



**Установка через скачивание**

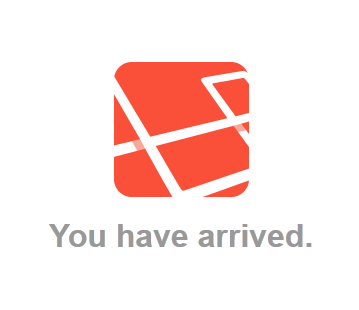
Сперва необходимо скачать сам архив по ссылке <https://github.com/laravel/laravel/archive/master.zip> и извлечь архив в папку на сервере. Дальше в корне приложения выполнить команду composer-а для установки всех зависимостей библиотеки.

|  |
| --- |
| **Установка зависимостей. Листинг 1.2** |
| composer installд |

**Запуск проекта**

Перейдем в папку с проектом: laraverl/public

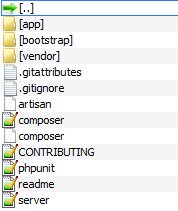
Должны увидеть следующее:



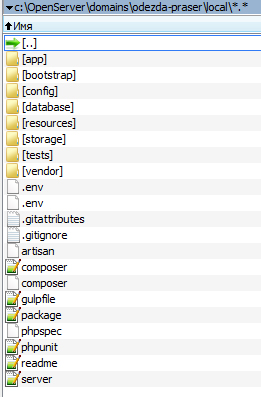
**4. Удаление public из url проекта**

Для того, чтобы к laravel-проекту можно было обращаться без вызова папки public необходимо сделать следующие действия:

1. Создать в корневой дирректории проекта папку local (или с любым другим именем).
2. Перенести все файлы и папки проекта, за исключением папки public в созданную папку local. Содержимое папки local получится примерно таким:



1. Содержимое папки public перенести в корень проекта. После чего саму папку public можно удалить. Корневая дирректория проекта будет выглядеть примерно так:



1. Сейчас необходимо изменить несколько путей. В корневом файле index.php изменим пути до файлов autoload.php и start.php.

|  |
| --- |
| **Новые пути до файлов autoload.php и start.php. Файл index.php. Листинг 1.2** |
| **require** \_\_DIR\_\_.'/local/bootstrap/autoload.php';  …  **$app** = require\_once \_\_DIR\_\_.'/local/bootstrap/start.php'; |

5. Настройка

Основные файлы настроек хранятся в app/config/. Каждой группе настроек отведен свой файл, либо папка.

С проектом можно работать одновременно локально и удаленно, не переписывая настройки, например подключения к базе данных, либо вывода ошибок. Именно поэтому, помимо стандартных файлов настройки, есть папка local, где для локальной машины можно переопределить часть настроек.

Получить имя текущей среды, можно с помощью команды:

|  |
| --- |
| **Имя текущей среды. Листинг 1.2** |
| $environment = **App::environment()**; |

По умолчанию, Laravel всегда использует среду Production. Настроить другие среды можно в файле bootstrap/start.php. В этом файле есть вызов

$app->detectEnvironment() — массив, который ему передаётся, используется для определения текущей среды.

6. Маршрутизация

Запрос из адресной строки попадает в так называемый обработчик маршрутов, или маршрутизатор, или роутер (routes). Маршрутизатор определяет какой контроллер необходимо вызывать.

Маршруты определяются в файле app/routes.php

|  |
| --- |
| **Простейший get-маршрут. Листинг 1.2** |
| Route::**get**('/', function () {  return 'Hello World';  }); |

Для перехвата POST-данных, можно воспользоваться методом Route::post

|  |
| --- |
| **Простейший post-маршрут. Листинг 1.2** |
| Route::**post**('foo/bar', function () {  return 'Hello World';  }); |

Метод Route::any перехватывает и POST и GET данные.

|  |
| --- |
| **Маршрут любого http-запроса. Листинг 1.2** |
| Route::**any**('foo', function () {  return 'Hello World';  }); |

Для перехвата маршрутов только по протоколу HTTPS, вторым входящим параметром можно передать не функцию, а массив, первым элементом которого является тип протокола, а вторым – функция.

|  |
| --- |
| **Маршрут любого https-запроса. Листинг 1.2** |
| Route::get('foo', array('**https**', function() {  return 'Must be over HTTPS';  })); |

Для генерации URL к какому-нибудь маршруту, можно воспользоваться методом URL::to(). Это может пригодится для генерации путей к ссылкам (значение атрибута href), картинкам (src), стилям (href), скриптам(src), обработчику форм (action) либо при переадресации.

|  |
| --- |
| **Генерация URL к маршруту. Листинг 1.2** |
| $url = **URL::to**('foo'); |

Для этих же целей можно использовать метод url

|  |
| --- |
| **. Листинг 1.** |
| $url = **url**('foo'); |

Параметры маршрутов

Добавление к маршруту обязательного параметра id.

|  |
| --- |
| **Добавление к маршруту обязательного параметра id. Листинг 1.2** |
| Route::get('user**/{id}**', function ($id) {  return 'User '.**$id**;  }); |

Добавление к маршруту необязательного параметра name.

|  |
| --- |
| **Добавление к маршруту необязательного параметра name. Листинг 1.2** |
| Route::get('user/**{name?}**', function (**$name = null**) {  return $name;  }); |

Вместо $name = null можно использовать любое значение по умолчанию.

|  |
| --- |
| **Добавление к маршруту необязательного параметра name. Листинг 1.2** |
| Route::get('user/**{name?}**', function (**$name =’Jhon’**) {  return $name;  }); |

Использование регулярных выражений в маршрутах.

|  |
| --- |
| **Маршруты с соответствием пути регулярному выражению. Листинг 1.2** |
| Route::get('user/{name}', function ($name) {  })->**where('name', '[A-Za-z]+')**;  Route::get('user/{id}', function ($id) {  })**->where('id', '[0-9]+');** |

Вместо последовательного вызова метода where, можно передать массив ограничений.

|  |
| --- |
| **Использование массива в регулярных вырражениях. Листинг 1.2** |
| Route::get('user/{name}', function ($name) {  })->**where('name', '[A-Za-z]+')**;  Route::get('user/{id}', function ($id) {  })**->where('id', '[0-9]+');** |

Именованные маршруты

Задать имя маршруту можно следующим способом:

|  |
| --- |
| **Назначение имени текущего исполняемого маршрута. Листинг 1.2** |
| Route::get('user/profile', array(**'as' => 'profile'**, function () {  //  })); |

Также можно задать контроллер и его экшн, который будет выполняется по данному маршруту.

|  |
| --- |
| **Назначение контроллера и экшна. Листинг 1.2** |
| Route::get('user/profile', array('as' => 'profile',  **'uses' => 'UserController@showProfile'**)); |

Теперь можно использовать имя маршрута при генерации URL, либо при перенаправлении.

|  |
| --- |
| **Генерация URL. Листинг 1.2** |
| $url = **URL::route('profile');**  $redirect = **Redirect::route('profile');** |

Получить имя текущего выполняемого маршрута можно методом currentRouteName():

|  |
| --- |
| **Получить имя текущего исполняемого маршрута. Листинг 1.2** |
| $name = Route::**currentRouteName**(); |

7. Middleware

HTTP Middleware (посредники) - это фильтры обработки HTTP-запроса. Так, например, в Laravel включены middlewares для проверки аутентификации пользователя. Если пользователь не залогинен, middleware перенаправляет его на страницу логина. Если же залогинен - middleware не вмешивается в прохождение запроса, передавая его дальше по цепочке middleware-посредников к собственно приложению.

Middlewares можно использовать в качестве фильтров для роутов.

Конечно, проверка авторизации - не единственная задача, которую способны выполнять middlewares. Это также добавление особых заголовков (например, CORS http-ответ вашего приложения) или логирование всех http-запросов.

В Laravel есть несколько дефолтных middleware, которые находятся в папке app/Http/Middleware. Это middlewares для реализации режима обслуживания сайта ("сайт временно не работает, зайдите позже"), проверки авторизации, CSRF-защиты и т.п.

Создание middleware

Для создания middleware можно воспользоваться командой make:middleware:

|  |
| --- |
| **Создание middleware. Листинг 1.** |
| php artisan make:middleware Admin |

Давайте создадим middleware Admin, который будет пропускать только админов, у которых в таблице users в базе данных выставлен параметр isAdmin = 1.

Middleware Admin сделаем по примеру Authenticate.

В папке app/Http/Middleware будет создан файл с классом Admin, к которому добавим следующую логику:

|  |
| --- |
| **. Листинг 1.** |
| if(Auth::user()->isAdmin != 1){  return redirect()->guest('auth/login');  } |

Чтобы пропустить запрос дальше, нужно вызвать функцию-замыкание $next с параметром $request.

Лучше всего представлять middlewares как набор уровней, которые HTTP-запрос должен пройти, прежде чем дойдёт до вашего приложения. На каждом уровне запрос может быть проверен по различным критериям и, если нужно, полностью отклонён.

Далее необходимо добавить middleware в свойство routeMiddleware класса app/Http/Kernel.php, назначив ему некоторое имя, например, admin, которое будет ключем массива:

|  |
| --- |
| **. Листинг 1.** |
| protected $routeMiddleware = [  'auth' => 'App\Http\Middleware\Authenticate',  **'admin' => 'App\Http\Middleware\Admin',**  'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',  ]; |

Теперь мы можем закрывать контроллеры от неавторизированных пользователей:

|  |
| --- |
| **Закрытие контроллеров от авторизированных пользователей. Листинг 1.** |
| public function \_\_construct()  {  parent::\_\_construct();  **$this->middleware('admin');**  } |

Если нужно, чтобы через middleware проходили все HTTP-запросы, то просто добавьте его в свойство $middleware класса app/Http/Kernel.php:

|  |
| --- |
| **. Листинг 1.** |
| protected $middleware = [  'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',  'Illuminate\Cookie\Middleware\EncryptCookies',  'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',  'Illuminate\Session\Middleware\StartSession',  'Illuminate\View\Middleware\ShareErrorsFromSession',  'App\Http\Middleware\VerifyCsrfToken',  ]; |

8. Пространства имен: основы

Имя класса может быть указано тремя способами:

* Неполные имена, такие как $a = new foo(); или foo::staticmethod();. Если текущее пространство имен currentnamespace, то эти имена преобразуются в currentnamespace\foo.
* Полные имена, такие как $a = new subnamespace\foo(); или subnamespace\foo::staticmethod();. Если текущее пространство имен currentnamespace, то эти имена преобразуются в currentnamespace\subnamespace\foo. Если код находится в глобальном пространстве имен, то имена преобразуются в subnamespace\foo.
* Абсолютные имена, или имена с предшествующим префиксом, обозначающим глобальное пространство. $a = new \currentnamespace\foo(); или \currentnamespace\foo::staticmethod();. Имена всегда определяются также как и записаны: currentnamespace\foo.

Разберемся с пространством имен на примере:

Создадим класс Foo

|  |
| --- |
| **Класс Foo. Листинг 1.** |
| class Foo  {  public function doAwesomeFooThings()  {  echo 'say Hi to the listeners';  }  } |

Создать новый объект этого класса просто:

|  |
| --- |
| **Новый объект класса Foo. Листинг 1.** |
| // use\_foo.php  require 'Foo.php';  $foo = **new** Foo(); |

Добавим пространство имен.

|  |
| --- |
| **Пространство имен для класса Foo. Листинг 1.** |
| **namespace** Acme\Tools;  class Foo  {  public function doAwesomeFooThings()  {  }  } |

Создать новый объект с помощью пространства имен можно так:

|  |
| --- |
| **Создание объекта с помощью пространства имен. Листинг 1.** |
| $foo = **new** \Acme\Tools\Foo(); |

Инструкция use может быть использована для доступа к классу.

|  |
| --- |
| **Использование инструкции use. Листинг 1.** |
| **use** \Acme\Tools\Foo;  $foo = new Foo(); |

Чтобы избавиться от длинных вызовов классов, можно добавить ярлык.

|  |
| --- |
| **Использование ярлыка в пространстве имен. Листинг 1.** |
| **use** Acme\Tools\Foo **as** SomeFooClass;  $foo = new SomeFooClass(); |

7. Контроллеры

Контроллеры хранятся в папке app/controllers/. Этот путь, в свою очередь определен в файле composer.json в настройке classmap.

Все контроллеры должны наследовать класс BaseController. Этот класс также может хранится в папке app/controllers и в него можно поместить общую логику для других контроллеров. BaseController расширяет базовый класс Controller.

Сперва определим маршрут для контроллера в файле app/routes.php.

|  |
| --- |
| **Определение маршрута для контроллера. Листинг 1.2** |
| Route::get('static', '**StaticController@index**'); |

Т.е., если пользователь в адресной строке набирает static, маршрутизатор находит контроллер StaticController и вызвает экшн index. Создадим в папке app/controllers файл StaticController:

|  |
| --- |
| **Простейший контроллер StaticController. Листинг 1.2** |
| class **StaticController** extends BaseController {  public function **index**()  {  echo 'Ok';  }  } |

Теперь, если в адресной строке браузера набираем static, на получаем ответ Ok

Создание RESTfull-контроллера через artisan

Сперва в консоли перейдем в папку с файлом artisan. Находясь в этой папке выполним следующую консольную команду.

|  |
| --- |
| **Создание RESTfull-контроллеров с помощью команды controller:make интерфейса artisan. Листинг 1.2** |
| **php** artisan controller:make NewsController |

Получим такой контроллер:

|  |
| --- |
| **RESTfull-контроллер. Листинг 1.2** |
| <?php  class NewsController extends \BaseController {  public function index()  {  //  }  public function create()  {  //  }  public function store()  {  //  }  public function show($id)  {  //  }  public function edit($id)  {  //  }  public function update($id)  {  //  }  public function destroy($id)  {  //  }  } |

Также необходимо зарегистрировать роут к этому контроллеру.

|  |
| --- |
| **Регистрация роута в файле app/routes.php. Листинг 1.2** |
| Route::resource('news', 'NewsController'); |

Такая регитрация дает множество маршрутов для обработки RESTfull экшнов контроллера News. Сам сгенерированный контроллер уже имеет методы-заглушки для каждого из этих маршрутов с комментариями, которые напоминают вам о том, какие типы запросов они обрабатывают.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Путь | Действие | Имя маршрута |
| GET | /resource | index | resource.index |
| GET | /resource/create | create | resource.create |
| POST | /resource | store | resource.store |
| GET | /resource/{id} | show | resource.show |
| GET | /resource/{id}/edit | edit | resource.edit |
| PUT/PATCH | /resource/{id} | update | resource.update |
| DELETE | /resource/{id} | destroy | resource.destroy |

Где resource – имя контроллера.

Чтобы создать только часть из возможных экшнов, можно воспользоваться настройками --only или --except :

|  |
| --- |
| **Создание части RESTfull-экшнов. Листинг 1.2** |
| php artisan controller:make NewsController **--only**=index,show  php artisan controller:make NewsController **--except**=index |

Этот же набор можно указать при регистрации маршрута:

|  |
| --- |
| **Создание маршрутов для группы экшнов. Листинг 1.2** |
| Route::resource('news', 'PhotoController',  array('**only**' => array('index', 'show')));  // либо:  Route::resource('news', 'PhotoController',  array('**except**' => array('create', 'store', 'update', 'delete'))); |

Обработка неопределенных методов

Можно определить «catch-all» метод, который будет вызываться для обработки запроса, когда в контроллере нет соответствующего метода. Он должен называться missingMethod и принимать массив параметров запроса в виде единственного своего аргумента.

|  |
| --- |
| **Создание части RESTfull-экшнов. Листинг 1.2** |
| public function **missingMethod**($parameters)  {  //  } |

Вложенная папка для контроллеров

Сперва просто создаем папки, где будут находиться контроллеры.

app/controllers

app/controllers/auth

app/controllers/adminka

...

В самих контроллерах ничего не меняем. Из корневой папки проекта, открываем консоль и выполняем следующую команду composer:

|  |
| --- |
| **Перестройка автозагрузки контроллеров. Листинг 1.2** |
| composer **dump-autoload** |

8. Шаблоны

По умолчанию, laravel работает с шаблонизатором blade. Шаблоны создаются в папке app/views и имеют расширение blade.php. Шаблоны подключаются в экшне через функцию view(), входящим параметром в который передается имя шаблона без расширения blade.php.

Сперва создадим в папке view папку layouts для хранения базовых шаблонов. В папке layouts создадим файл defaults.blade.php

|  |
| --- |
| **Базовый шаблон defaults.blade.php. Листинг 1.2** |
| **@include('layouts.header')**  <div class="container">  <div class="masthead">  <h3 class="text-muted">Project name</h3>  <ul class="nav nav-justified">  <li class="active"><a href="#">Home</a></li>  --блок ссылок --  </ul>  </div>  **@yield('content')**    <!-- Site footer -->  <div class="footer">  <p>&copy; Company 2014</p>  </div>  </div> <!-- /container -->  **@include('layouts.footer')** |

В файлах layouts/header.blade.php и layouts/footer.blade.php находится обычный html-код для шапки и футера сайта. Эти 3 файла – это неизменная часть шаблона.

@yield(‘content’) – вывод переменной content. Саму переменную определим в меняющейся части шаблона.

Меняющуюся часть шаблона вынесем в отдельный фай л index.blade.php.

|  |
| --- |
| **index.blade.php. Листинг 1.2** |
| **@extends('layouts.default')**  **@section('content')**  <h1>Добро пожаловать на сайт</h1>  <div>Текст на страницу</div>  **@stop** |

Подключение шаблона index.blade.php осуществляется в экшне контроллера.

|  |
| --- |
| **Подключение подшаблона index.blade.php. Листинг 1.2** |
| return **view('index'**); |

Передача массива в шаблон:

|  |
| --- |
| **Передача массива в шаблон. Листинг 1.2** |
| $posts = array(1=>’One’, 2=>’Two’);  return **view('index', $posts**); |

Имеется также ещё один способ передачи переменных в шаблон, через метод with():

|  |
| --- |
| **Передача переменной в шаблон с помощью метода with. Листинг 1.** |
| $posts = array(1=>’One’, 2=>’Two’);  return view(‘index’)**->with**(‘posts’, $posts) |

Если вы не уверены в существовании передаваемой переменной, то нужно испльзовать with(), тогда не будет выводиться ошибка.

И ещё один способ передачи переменных:

|  |
| --- |
| **Магический метод передачи данных в шаблон. Листинг 1.** |
| $view = view('greeting')**->withName**('Victoria'); |

Вывод всех элементов массива на экран

|  |
| --- |
| **Вывод элементов массива на экран. Листинг 1.2** |
| @if($posts->count())  **@foreach($posts as $post)**  <div>{{$post }}</div>  **@endforeach**  @endif |

Передача данных во все шаблоны

Для передачи переменных во все шаблоны, можно воспользоваться либо хелперами:

|  |
| --- |
| **Передача данных во все шаблоны с помощью хелперов. Листинг 1.** |
| **view()->share**('data', [1, 2, 3]); |

либо фасадами:

|  |
| --- |
| **Передача данных во все шаблоны с помощью фасадов. Листинг 1.** |
| **View::share**('data', [1, 2, 3]); |

Этот код можно положить в метод boot() сервис-провайдера общего сервис-провайдера приложения AppServiceProvider или своего собственного.

View composer

Композеры (view composers) - функции-замыкания или методы класса, которые вызываются, когда шаблон рендерится в строку. Если у вас есть данные, которые вы хотите привязать к шаблону при каждом его рендеринге, то композеры помогут вам выделить такую логику в отдельное место.

Регистрировать композеры можно внутри сервис-провайдера. Мы будем использовать фасад View для того, чтобы получить доступ к имплементации контракта Illuminate\Contracts\View\Factory:

|  |
| --- |
| **Определение композера. Листинг 1.** |
| <?php namespace App\Providers;  use View;  use Illuminate\Support\ServiceProvider;  class ComposerServiceProvider extends ServiceProvider {  public function boot()  {  // Если композер реализуется при помощи класса:  **View::composer**('profile',  'App\Http\ViewComposers\ProfileComposer');  // Если композер реализуется в функции-замыкании:  **View::composer**('dashboard', function()  {  });  }  } |

Допустим, мы хотим, чтобы композер profile был реализован в классе (см. код выше). Метод ProfileComposer@compose будет вызываться каждый раз перед тем, как шаблон profile будет рендерится в строку. Давайте определим класс композера:

|  |
| --- |
| **Класс композера. Листинг 1.** |
| <?php namespace App\Http\Composers;  use Illuminate\Contracts\View\View;  use Illuminate\Users\Repository as UserRepository;  class ProfileComposer {  protected $users;  public function \_\_construct(UserRepository $users)  {  // Зависимости разрешаются автоматически службой контейнера...  $this->users = $users;  }  public function **compose(View $view)**  {  **$view->with**('count', $this->users->count());  }  } |

Метод compose должен получать в качестве аргумента инстанс Illuminate\Contracts\View\View. Для передачи переменных в шаблон используйте метод with().

Назначение композера для нескольких шаблонов

Вместо имени шаблона можно использовать массив имен.

|  |
| --- |
| **Назначение композера для нескольких шаблонов. Листинг 1.** |
| View::composer(['profile', 'dashboard'], 'App\Http\ViewComposers\MyViewComposer'); |

Композер для всех шаблонов

А вот так можно назначить композер для всех шаблонов:

|  |
| --- |
| **Назначение композера для всех шаблонов. Листинг 1.** |
| View::composer(**'\*'**, function()  {  //  }); |

Регистрация нескольких шаблонов

Можно использовать метод composers чтобы зарегистрировать несколько композеров одновременно:

|  |
| --- |
| **Регистрация нескольких шаблонов одновременно . Листинг 1.** |
| **View::composers**([  'App\Http\ViewComposers\AdminComposer' =>  ['admin.index', 'admin.profile'],  'App\Http\ViewComposers\UserComposer' => 'user',  'App\Http\ViewComposers\ProductComposer' => 'product'  ]); |

View Creator

Создатели шаблонов работают почти так же, как композеры, но вызываются сразу после создания объекта шаблона, а не во время его рендеринга в строку. Для регистрации используйте метод creator:

|  |
| --- |
| **Регистрация view creator. Листинг 1.** |
| **View::creator**('profile', 'App\Http\ViewCreators\ProfileCreator'); |

9. База данных MySQL, миграции

Подключение и настройка базы данных осуществляется в файле application/config/database.php. Для подключения нужной базы данных необходимо прописать настройки сервера баз данных в массив mysql:

|  |
| --- |
| **Настройка базы данных, элемент массива mysql. Листинг 1.2** |
| 'mysql' => array(  'driver' => 'mysql',  'host' => 'localhost',  **'database' => 'project',**  **'username' => 'root',**  **'password' => '',**  'charset' => 'utf8',  'collation' => 'utf8\_unicode\_ci',  'prefix' => '',  ), |

Выделенные жирным значения необходимо заменить на свои.

Миграции

Миграции базы данных являются весьма полезны для любого проекта, особенно для проектов с несколькими разработчиками, позволяя иметь последнюю версию базы данных у всех разработчиков. В Laravel для этого достатчно выполнить одну команду в командной строке.

Для создания новой миграции нам понадобится интерфейс командной строки Laravel — «Artisan».

Итак, откроем консоль командной строки из папки, где расположен файл artisan. В консоли введем следующую команду:

|  |
| --- |
| **Создание файла миграции. Листинг 1.2** |
| php **artisan migrate:make create\_categories** |

Консоль должна ответить нам следующей фразой:

“Great! New migration created!”

Если получили такой ответ, то перейдем в папку application/migrations. Там должен находится файл 2014\_04\_20\_210359\_create\_users.php (к имени миграции artisan добавляет текущую дату). Откроем данный файл. Увидим следующее:

|  |
| --- |
| **Файл миграции. Листинг 1.2** |
| class Create\_Categories {  /\*\*  \* Внести изменения в базу данных.  \*  \* @return void  \*/  public function **up()**  {  //  }  /\*\*  \* Отменить изменения базы данных.  \*  \* @return void  \*/  public function **down()**  {  //  }  } |

Класс миграции содержит два метода up() для внесения изменений в таблицу базы данных и down() для отменты действий метода up(). Например, если мы создаем таблицу в up(), то в down() ее нужно удалить.

Допишем действия up() и down()

|  |
| --- |
| **Создание таблицы и удаление в методах up() и down() файла миграции. Листинг 1.2** |
| public function up()  {  **Schema::create**('**categories**', function ($table) {  // auto incremental id (PK)  $table->increments('id');  // varchar 32  $table->string('name', 32);  $table->enum('showhide', array('show','hide'))->index();  // created\_at | updated\_at DATETIME  $table->timestamps();  });  Schema::table('categories', function(Blueprint $table)  {  //  });  }  public function down()  {  **Schema::drop('categories');**  } |

Внутри функции мы можем использовать следующие красивые методы для определения структуры таблицы:

* increments() — добавить автоинкрементируемое поле — его будет иметь бо?льшая часть ваших таблиц
* string() — создать поле VARCHAR — правда, «строка» куда более описательное имя, чем в стандарте SQL?
* integer() — добавить целочисленное поле
* float() — поле с дробным числом (число с плавающей точкой)
* boolean() — логическое («булево») поле — истина (true) или ложь (false)
* date() — поле даты
* timestamp() — поле «отпечатка времени», так называемый «Unix timestamp»
* text() — текстовое поле без ограничения по длине
* blob() — большой двоичный объект (BLOB)

Перед тем как на основе существующих миграций, создавать таблицы, давайте создадим таблицу migrations, в которой laravel будет хранить данные о самих миграциях:

|  |
| --- |
| **Создание таблицы миграций. Листинг 1.2** |
| // ВНИМАНИЕ! Данну команду необходимо выполнять, если у нас еще не создана таблица migrations  php artisan **migrate:install** |

В результате мы должны увидеть

“Migration table created successfully.”

Теперь, когда таблица создана, мы можем выполнить саму миграцию:

|  |
| --- |
| **Выполнение миграций. Листинг 1.2** |
| php artisan **migrate** |

Чтобы удалить таблицу, можно выполнить команду rollback

|  |
| --- |
| **Выполнение миграций. Листинг 1.2** |
| php artisan **migrate:rollback** |

Запросы к базе данных, CRUD-операции

SELECT

Для создания запросов SELECT-запросов, сперва необходимо создать объект класса DB и статического метода table(), входящим параметром в который необходимо передать имя таблицы.

|  |
| --- |
| **Создание объекта запроса. Листинг 1.2** |
| $cat = **DB::table**(‘categories’); |

Для получения результата можно воспользоваться следующими методами:

* get() его мы только что использовали, он возвращает массив объектов — результирующих строк с полями;
* first() этот метод вернёт один объект-результат, который подошёл под критерии запроса;
* find($id) метод находит запись по её ID; это краткая форма для записи where('id', '=', $id); возвращает один объект-результат;
* only($fieldname) возвра0щает значение одного поля, подходящего под запрос;
* get(array(…)) мы можем передать методу get() массив полей, чтобы получить только их.

Получить все поля таблицы можно используя метод get().

|  |
| --- |
| **Создание объекта запроса. Листинг 1.2** |
| $cat = DB::table(‘categories’)->**get()**; |

Уточнение запроса осуществляется с помощью метода where()

|  |
| --- |
| **Применение метода where(). Листинг 1.2** |
| $cat = DB::table(‘categories’)**->where(‘showhide’, ‘=’, ‘show’)**->get(); |

Конструктор запросов может быть использован для выборки данных из нескольких таблиц через JOIN.

|  |
| --- |
| **Выборка данных из нескольких таблиц через Join. Листинг 1.** |
| DB::table('users')  ->join('contacts', 'users.id', '=', 'contacts.user\_id')  ->join('orders', 'users.id', '=', 'orders.user\_id')  ->select('users.id', 'contacts.phone', 'orders.price'); |

Если необходимо сделать сложную фильтацию, то в значение методов where() или orWhere() можно добавлять функуию с дополнительным query-запросом:

|  |
| --- |
| **Сложная фильтрация . Листинг 1.** |
| DB::table('users')  ->where('name', '=', 'Джон')  ->**orWhere(function ($query)** {  $query->where('votes', '>', 100)  ->where('title', '<>', 'Админ');  })->get(); |

INSERT

|  |
| --- |
| **Вставка данных . Листинг 1.** |
| DB::table('users')**->insert**(  array('email' => 'john@example.com', 'votes' => 0)  ); |

Если необходимо вставить данные и при этом тут же получить id, можно воспользоваться методом insertGetId.

|  |
| --- |
| **Вставка данных и получение id. Листинг 1.** |
| $id = DB::table('users')->**insertGetId**(  array('email' => 'john@example.com', 'votes' => 0)  ); |

Laravel также поддерживает множественную вставку данных:

|  |
| --- |
| **Множественная вставка данных. Листинг 1.** |
| DB::table('users')->insert(array(  array('email' => 'taylor@example.com', 'votes' => 0),  array('email' => 'dayle@example.com', 'votes' => 0),  )); |

UPDATE

|  |
| --- |
| **Обновление данных. Листинг 1.** |
| DB::table('users')  **->where('id', 1)**  ->update(array('votes' => 1)); |

DELETE

|  |
| --- |
| **Удаление . Листинг 1.** |
| DB::table('users')->where('votes', '<', 100)->**delete**(); |

10. Модели

Модели создаем в папке models. Имя файла и класса модели laravel ассоциирует с именем таблицы, причем из таблицы убирается окончание “s”. Таким образом мы создали модель для таблицы products.

|  |
| --- |
| **Простейшая модель . Листинг 1.** |
| <?php  class Product extends **Eloquent**  {    } |

Eloquent - класс ORM для модели, который используется в Laravel. Он обеспечивает работу с объектами базы данных.

Свойство $timestamps получает значение true, что указывает на автоматическое обновление полей created\_at и updated\_at при создании или обновлении записи соответственно.

Внимание: Eloquent предполагает, что каждая таблица имеет первичный ключ с именем id. Вы можете определить свойство $primaryKey для изменения этого имени. Аналогичным образом, вы можете определить свойство $connection для задания имени подключения к БД, которое должно использоваться при работе с данной моделью.

Кроме свойства $primaryKey Laravel предопределяет следующие свойства класса модели:

|  |
| --- |
| **Предопределенные свойства модели. Листинг 1.** |
| <?php  class Product extends **Eloquent**  {  protected **$table** = 'my\_users';  public static **$timestamps** = true;  protected **$fillable** = array('username, 'email'); //указание на то, какие поля могут массово назначатся  protected **$guarded** = array('id', 'password'); // запрет на массовое назначение перечисленных полей.  protected **$hidden** = array('password'); //позволяет скрыть определенные столбцы при возвращении экземпляра модели  } |

Массовое заполнение

При создании новой модели вы передаёте её конструктору массив атрибутов. Эти атрибуты затем присваиваются модели через массовое заполнение. Это удобно, но в то же время представляет серьёзную проблему с безопасностью, когда вы передаёте ввод от клиента в модель без проверок — в этом случае пользователь может изменить любое и каждое поле вашей модели. По этой причине по умолчанию Eloquent защищает вас от массового заполнения.

Методы возврата данных

Модель включает три простых метода для возврата конкретных элементов данных из неё:

|  |
| --- |
| **Методы возврата данных. Листинг 1.** |
| public function **getAuthIdentifier()**  {  return $this->getKey(); // вернет id пользователя  }  public function **getAuthPassword()**  {  return $this->password; // вернет пароль пользователя  }  public function **getReminderEmail()**  {  return $this->email; // вернет email для отправки напоминания о пароле  }  protected function **getDateFormat()**  {  return 'U'; // использование собственного формата времени  } |

Как только модель определена, у вас всё готово для того, чтобы можно было выбирать и создавать записи. Обратите внимание, что вам нужно создать в этой таблице поля updated\_at и created\_at. Если вы не хотите, чтобы они были автоматически используемы, установите свойство $timestamps класса модели в false.

|  |
| --- |
| **Получение всех записей модели. Листинг 1.** |
| $users = **User::all()**; |

По первичному ключу записи можно получить так:

|  |
| --- |
| **Получение записи по первичному ключу. Листинг 1.** |
| $user **= User::find(1);**  var\_dump($user->name); |

Все методы, доступные в конструкторе запросов, также доступны в запросах с использованием моделей.

Иногда вам нужно возбудить исключение, если определённое значение не найдено, что позволит вам его отловить в обработчике App::error() и вывести страницу 404 («Не найдено»).

|  |
| --- |
| **Получение данных по первичному ключу с возбуждением исключения. Листинг 1.** |
| $model = User::**findOrFail**(1);  $model = User::where('votes', '>', 100)->**firstOrFail**(); |

Для регистрации обработчика ошибки необходимо подписаться на событие ModelNotFoundException:

|  |
| --- |
| **. Листинг 1.** |
| use Illuminate\Database\Eloquent\**ModelNotFoundException**;  A**pp::error**(function (ModelNotFoundException $e) {  return Response::make('Not Found', 404);  }); |

Вставка

Данные через модель вставляются следующим образом:

|  |
| --- |
| **Вставка данных. Листинг 1.** |
| $user = **new User**; // создаем объект  $user->name = 'Джон'; // данные  $user->**save()**; // сохранение |

Внимание: обычно модели Eloquent содержат автоматические числовые ключи (autoincrementing). Однако если вы хотите использовать собственные ключи, установите свойство $incrementing класса модели в значение false.

Можно также использовать метод create() для создания и сохранения модели одной строкой. Метод вернёт добавленную модель. Однако перед этим нужно определить либо свойство $fillable, либо $guarded в классе модели, так как изначально все модели Eloquent защищены от массового заполнения.

|  |
| --- |
| **Всавка данных путем массового заполнения . Листинг 1.** |
| $user = User::**create**(array('name' => 'Джон')); |

Обновление

|  |
| --- |
| **Обновление данных по id. Листинг 1.** |
| $user = **User::find(1);**  $user->email = 'alex@ya.ru';  $user->save(); |

Удаление

|  |
| --- |
| **Удаление данных по id. Листинг 1.** |
| $user = **User::find(1);**  $user->**delete();** |

Заготовки позволяют повторно использовать логику запросов в моделях. Для создания заготовки просто начните имя метода со scope:

|  |
| --- |
| **Заготовки запросов. Листинг 1.** |
| class User extends Eloquent {  public function **scopePopular**($query)  {  return $query->where('votes', '>', 100);  }  public function **scopeWomen**($query)  {  return $query->whereGender('W');  }  } |

Использование заготовок запросов

|  |
| --- |
| **Использование заготовок запросов. Листинг 1.** |
| $users = **User::popular()->women()**->orderBy('created\_at')->get(); |

Можно также использовать заготовки запросов с входящими параметрами:

|  |
| --- |
| **Входящие параметры в заготовках запросов. Листинг 1.** |
| class User extends Eloquent {  public function scopeOfType($query, $type)  {  return $query->**whereType($type)**;  }  } |

Вызов заготовок запросов с входящими параметрами:

|  |
| --- |
| **Входящие параметры в заготовках запросов. Листинг 1.** |
| $users = User::**ofType('member')**->get(); |

Связи в моделях

Логика связей в Laravel осуществляется с помощью класса Eloquent. Eloquent упрощает работу и управление такими отношениями, и поддерживает следующие типы связей:

Один к одному

Один ко многим

Многие ко многим

Ко многим через

Полиморфические связи

Полиморфические связи многие ко многим

Один к одному / hasOne

Связь вида «один к одному» является очень простой. К примеру, модель User может иметь один Phone. Мы можем определить такое отношение в Eloquent:

|  |
| --- |
| **Связь hasOne. Листинг 1.** |
| class User extends Eloquent {  public function phone()  {  return **$this->hasOne('Phone');**  }  } |

loquent считает, что поле в таблице называется по имени модели плюс \_id. В данном случае предполагается, что это user\_id. Если мы хотим перекрыть стандартное имя необходимо передать второй параметр методу hasOne(). Кроме того мы можем передать в метод третий аргумент, чтобы указать, какие локальные столбцы следует использовать для объединения:

|  |
| --- |
| **Передача дополнительных аргументов связи hasOne. Листинг 1.** |
| return $this->hasOne('Phone', **'foreign\_ke**y');  return $this->hasOne('Phone', 'foreign\_key', **'local\_key'**); |

Принадлежность / belongsTo

Для создания обратного отношения в модели имеется специальный метод belongsTo() («принадлежит к»):

|  |
| --- |
| **Связь belongsTo. Листинг 1.** |
| class Phone extends Eloquent {  public function user()  {  return **$this->belongsTo('User')**;  }  } |

В примере выше Eloquent будет искать поле user\_id в таблице phones. Если вы хотите назвать внешний ключ по другому, передайте это имя вторым параметром к методу belongsTo():

|  |
| --- |
| **Передача аргумента внешнего ключа связи belongsTo. Листинг 1.** |
| class Phone extends Eloquent {  public function user()  {  return $this->belongsTo('User', **'local\_key'**);  }  } |

Кроме того, можно передать третий аргумент, который определить имя связного столбца в родительской таблице.

|  |
| --- |
| **Передача аргумента для определения связного столбца в родительской таблице. Листинг 1.** |
| class Phone extends Eloquent {  public function user()  {  return $this->belongsTo('User', 'local\_key', **'parent\_key'**);  }  } |

Один ко многим / hasMany

Примером отношения «один ко многим» является статья в блоге, которая имеет «много» комментариев. Мы можем смоделировать это отношение таким образом:

|  |
| --- |
| **Связь hasMany. Листинг 1.** |
| class Post extends Eloquent {  public function comments()  {  return **$this->hasMany('Comment');**  }  } |

Теперь мы можем получить все комментарии с помощью динамического свойства:

|  |
| --- |
| **Получение данных из связующей модели. Листинг 1.** |
| $comments = Post::find(1)->comments; |

Если нужно добавить ограничения на получаемые комментарии, можно вызвать метод comments() и продолжить добавлять условия:

|  |
| --- |
| **Добавление условий к связующей таблице. Листинг 1.** |
| $comments = Post::find(1)->comments()**->where**('title', '=', 'foo')->first(); |

Многие ко многим / belongsToMany

Отношения типа «многие ко многим» — более сложные, чем остальные виды отношений. Примером может служить пользователь, имеющий много ролей, где роли также относятся ко многим пользователям. Например, один пользователь может иметь роль admin. Нужны три таблицы для этой связи: users, roles и role\_user. Название таблицы role\_user происходит от упорядоченных по алфавиту имён связанных моделей и она должна иметь поля user\_id и role\_id.

|  |
| --- |
| **Связь belongsToMany. Листинг 1.** |
| class User extends Eloquent {  public function roles()  {  return $this->**belongsToMany('Role');**  }  } |

Теперь мы можем получить роли пользователей следующим образом:

|  |
| --- |
| **Получение ролей пользователя. Листинг 1.** |
| $roles = User::find(1)**->roles**; |

Полиморфические отношения

Полиморфические отношения позволяют модели быть связанной с более, чем одной моделью.

11. Модуль авторизации

В модуль авторизации входят экшны и шаблоны регистрации, авторизации, выхода и проверки.

Сперва создадим rout авторизации

|  |
| --- |
| **Маршруты для методов авторизации. Листинг 1.2** |
| Route::**get**('/**login**', [  'as' => '**user-login**',  'uses' => 'AuthController@**getLogin**'  ]);  Route::**get**('/**reg**', [  'as' => '**user-reg**',  'uses' => 'AuthController@**getReg**'  ]);  //post  Route::**post**('/**login**', [  'uses' => 'AuthController@**postLogin**'  ]);  Route::**post**('/**reg**', [  'uses' => 'AuthController@**postReg**'  ]); |

Или так:

|  |
| --- |
| **Маршрут для контроллера авторизации. Листинг 1.** |
| Route::controller(‘auth’, AuthController); |

Создадим таблицу для хранения пользовательских данных:

|  |
| --- |
| **Метод up файла миграции users . Листинг 1.** |
| public function up()  {  Schema::create('users', function(Blueprint $table) {  // ID пользователя  $table->increments('id');    // E-Mail (уникальный)  $table->string('email')->unique();    // Пароль. Для используемой в Laravel хэш-функции требуется не меньше 60 символов  $table->string('password', 60);    // Никнейм  $table->string('username')->unique();    // Админ?  $table->boolean('isAdmin');    // Активирован?  $table->boolean('isActive')->index();    // Код активации аккаунта  $table->string('activationCode');    // Токен для возможности запоминания пользователя  $table->rememberToken(); // remember\_token    // created\_at, updated\_at  $table->timestamps();  });  } |

Если мы используем таблицу users, то саму модель делать не надо, т.к. в папке models уже есть файл модели User.php.

Контроллер auth содержит следующие методы:

|  |
| --- |
| **Контроллер auth. Листинг 1.2** |
| class AuthController extends BaseController {  public function **getReg**(){  return View::make('auth.reg');  }  public function **getLogin**(){  return View::make('auth.login');  }  public function **postReg**(){  //логика регистрации  }  public function **postLogin**(){  //логика авторизации  }  public function **getLogout**(){  if (Auth::check()){  Auth::logout();  return Redirect::route('user-login')  ->withErrors('Вы разлогинились!');  }  }  } |

Файл шаблона auth/reg

|  |
| --- |
| **Файл шаблона reg. Листинг 1.2** |
| @extends('layouts.default')  @section('content')  <div class="catalog\_odejdy">  <h1>Авторизация</h1>  @foreach ($errors->all() as $error)  <div class="errors">{{ $error }}</div>  @endforeach    {{ **Form::open(array('route' => 'user-reg'))**; }}  {{ Form::text('username', Input::old('username'), array('placeholder'=>'login'))}}<br />  {{ Form::text('email', Input::old('email'), array('placeholder'=>'E-mail'));}}<br />  {{ Form::password('password', '', array('placeholder'=>'пароль'));}}<br />  {{Form::submit('Регистрация!');}}  {{ SPecialiST RePack}}  </div>  <div class="clear"></div> |

Данные из формы отправляются на маршрут с именем user-reg, который запускает экшн postReg() контроллера Auth.

|  |
| --- |
| **Экшн postReg контроллера Auth. Листинг 1.2** |
| public function postReg(){  $data = Input::all();  $rules = array(  '**username**' => array('required', **'unique:users,username'**),  '**password**' => array('required', 'min:7')  );  **$validation = Validator::make($data, $rules);**  if ( **$validation->fails() )** {  $errors = **$validation->messages();**  }  **$user = new User();**  $user->username = Input::get('username');  $user->email = Input::get('email');  $user->password = Hash::make(Input::get('password'));  $user->activationCode = Input::get('password');  $user->save();  return Redirect::route('user-login');  } |
|  |

Рассмотрим экшн postLogin

|  |
| --- |
| **Экшн postLogin . Листинг 1.** |
| public function postIndex(){  $data = array(  'username' => Input::get('username'),  'password' => Input::get('password'),  );  **$auth = Auth::attempt($data, false);**  if(! $auth){  return Redirect::to('auth')->withErrors(array(  'Ошибка авторизации'  ));  }  return Redirect::to('/cabinet');  } |

Как видно из листинга, вся логика авторизации осуществлена в статичном методе attempt класса Auth, куда необходимо передать массив с данными.

В шаблонах и контроллерах можно делать проверку if(Auth::check), которая будет проверять не авторизирован ли пользователь:

|  |
| --- |
| **Проверка авторизирован ли пользователь. Листинг 1.2** |
| **@if(Auth::check())**  <div align='right'>{{ HTML::link('logout', 'Выход') }}</div>  **@endif** |

Данные об авторизированном пользователе можно получить так:

|  |
| --- |
| **Данные авторизированного пользователя. Листинг 1.** |
| echo **Auth::user()**->email; |

Для групп маршрутов авторизации применим фильтр auth:

|  |
| --- |
| **Применение фильтра auth для групп маршрутов. Листинг 1.2** |
| //adminka  Route::group(**array('before' => 'auth')**, function () {  Route::get('/adminka',[  'as'=>'adminka',  'uses'=>'AdminController@index'  ]);  Route::get('user/profile', function () {  // К этому маршруту также привязан фильтр auth.  });  }); |

Переопределить redirect после авторизации, можно в контроллере Auth:

|  |
| --- |
| **Переопределение редиректа после авторизации. Листинг 1.** |
| protected **$redirectTo** = ‘/’ |

12. Постраничная навигация

Применение постраничной навигации в конструкторе запросов

|  |
| --- |
| **Постраничная навигация в конструкторе запросов. Листинг 1.** |
| $users = DB::table('users')->**paginate**(15); |

Постраничная навигация в ORM-моделях:

|  |
| --- |
| **Постраничная навигация в ORM-моделях . Листинг 1.** |
| $users = User::paginate(15);  $users = User::where('votes', '>', 100)->paginate(15); |

Вывод ссылок постраничной навигации в шаблонах:

|  |
| --- |
| **Вывод постраничной навигации в шаблонах. Листинг 1.** |
| {{$users->links()}} |

13. Модуль обработки изображений

Для изменения размеров изображений можно использовать модуль Image (http://image.intervention.io/getting\_started/installation ), который необходимо установить через composer.

|  |
| --- |
| **Установка модуля Image. Листинг 1.** |
| Composer require intervention/image |

Далее открываем файл config/app.js, в элемент массива $providers добавим такую строчку:

|  |
| --- |
| **Изменение массива $providers. Листинг 1.** |
| 'Intervention\Image\ImageServiceProvider' |

Также, в этом файле находим массив $aliases, к которому необходимо добавить еще один элемент:

|  |
| --- |
| **Добавление элемента массива к массиву $aliases. Листинг 1.** |
| 'Image' => 'Intervention\Image\Facades\Image' |

Теперь у нас будет доступен класс Image.

|  |
| --- |
| **Уменьшение изображение с сохранением пропорций. Листинг 1.** |
| **$img = Image::make**(‘путь к оригинальному изображению’);  $img->resize(235, **null, function($constraint) {**  **$constraint->aspectRatio();**  **});**  $pic\_small = 's\_'.$filename;  $img->save('путь к папке/'.$pic\_small); |

14. Profiler Debugbar

Сперва пропишем дополнительную зависимость в файл composer.json

|  |
| --- |
| **Добавляем зависимость профайлера. Листинг 1.** |
| "require": {  "laravel/framework": "4.2.\*",  **"barryvdh/laravel-debugbar":"\*"**  }, |

Далее открываем файл config/app.php.

К массиву providers добавляем следующий элемент:

|  |
| --- |
| **Новый элемент массива providers. Листинг 1.** |
| 'Barryvdh\Debugbar\ServiceProvider', |

К массиву aliases также добавим один элемент:

|  |
| --- |
| **Новый элемент массива aliases. Листинг 1.** |
| 'Debugbar' => 'Barryvdh\Debugbar\Facade', |

Далее открываем консоль и обновляем зависимости через composer.

|  |
| --- |
| **Обновление зависимостей. Листинг 1.** |
| Composer update |

Осталось в файле app.php включить профайлер, выставив значение элемента массива.

|  |
| --- |
| **Включаем профайлер. Листинг 1.** |
| ‘debug’=>true |

15. Artisan

Artisan - название интерфейса командной строки, входящей в состав Laravel. Он предоставляет полезные команды для использования во время разработки вашего приложения. Работает на основе мощного компонента Symfony Console.

Каждая команда также включает и инструкцию, которая отображает и описывает доступные аргументы и опции для команды. Чтобы её вывести, необходимо добавить слово help перед командой:

php artisan help migrate

Вы также можете указать среду, в которой будет выполнена команда при помощи опции --env:

php artisan migrate --env=local

Вы также можете увидеть версию Laravel вашего приложения используя опцию --version:

php artisan –version

Доступные команды:

php artisan list — вывод всех доступных команд

make:command — создаёт новый класс команды

make:console — создаёт новую команду Artisan

make:controller — создаёт новый класс контроллера ресурса

make:event — создаёт новый класс события

make:middleware — создаёт новый класс промежуточного ПО

make:migration — создаёт новый файл миграции

make:model — создаёт новый класс модели Eloquent

make:provider — создаёт новый класс поставщика услуг

make:request — создаёт новый класс запроса формы

event:generate — генерирует пропущенные события и обработчики

В дополнение к командам, предоставляемых Artisan-ом, можно создавать свои собственные команды для работы с вашим приложением. Свои команды можно хранить как в директории app/Console, так и самостоятельно выбирать место для хранения, прежде убедившись, что команды будут автоматически загружены, основываясь на настройках composer.json.

Создание команды

Для создания новой команды, можно воспользоваться командой Artisan-а make:console, которая сгенерируетмакет класса:

|  |
| --- |
| **Генерация нового класса команды. Листинг 1.** |
| php artisan make:console FooCommand |

Команда выше сгенерирует класс app/Console/FooCommand.php.

Создавая команду, опция --command может быть использована для назначения имени команды в консоли:

|  |
| --- |
| **Назначение имени команды в консоли. Листинг 1.** |
| php artisan make:console AssignUsers --command=users:assign |

Написание команды

Как только команда будет сгенерирована, необходимо заполнить свойства класса name и description, которые будут использованы при отображении команды в списке.

Метод fire будет вызван как только ваша команда будет запущена. Вы можете поместить в этот метод любую логику.

Аргументы и опции

В методах getArguments и getOptions вы можете определить любые аргументы или опции, которые будет принимать команда. Оба этих метода возвращают массив команд, описываемых списком полей массива.

|  |
| --- |
| **Массив, определяющий аргументы. Листинг 1.** |
| array($name, $mode, $description, $defaultValue) |

Аргумент mode может принимать одно из следующих значений: InputArgument::REQUIRED (обязательный) или InputArgument::OPTIONAL (необязательный).

|  |
| --- |
| **Массив, определяющий опции. Листинг 1.** |
| array($name, $shortcut, $mode, $description, $defaultValue) |

Для опций, аргумент mode может быть: InputOption::VALUE\_REQUIRED (значение обязательно), InputOption::VALUE\_OPTIONAL (значение необязательно), InputOption::VALUE\_IS\_ARRAY (значение - это массив), InputOption::VALUE\_NONE (нет значения).

Режим VALUE\_IS\_ARRAY обозначает, что этот переключатель может быть использован несколько раз при вызове команды:

|  |
| --- |
| **Использование режима VALUE\_IS\_ARRAY. Листинг 1.** |
| php artisan foo --option=bar --option=baz |

Значение VALUE\_NONE означает, что опция просто используется как "переключатель":

|  |
| --- |
| **. Листинг 1.** |
| php artisan foo --option |

Во время выполнения команды, очевидно, потребуется получать значения переданных аргументов и опций. Для этого можно воспользоваться методами argument и option:

|  |
| --- |
| **Получение ввода. Листинг 1.** |
| //Получение значения аргумента команды  $value = $this->**argument**('name');  //Получение всех аргументов  $arguments = $this->argument();  //Получение значения опции команды  $value = $this->**option**('name');  //Получение всех опций  $options = $this->option(); |

Для вывода данных в консоль вы можете использовать методы info (информация), comment (комментарий), question (вопрос) и error (ошибка). Каждый из этих методов будет использовать цвет по стандарту ANSI, соответствующий смыслу метода.

|  |
| --- |
| **Команды вывода. Листинг 1.** |
| //Вывод информации в консоль  $this->**info**('Display this on the screen');  //Вывод сообщений об ошибке в консоль  $this->**error**('Something went wrong!');  //Взаимодействие с пользователем  //Вы также можете воспользоваться методами ask и confirm для обеспечения пользовательского ввода:  //Попросить пользователя ввести данные:  $name = $this->**ask**('What is your name?');  //Попросить пользователя ввести секретные данные:  $password = $this->**secret**('What is the password?');  //Попросить пользователя подтвердить что-то:  if ($this->**confirm**('Do you wish to continue? [yes|no]'))  {  //  }  //Вы также можете указать ответ по умолчанию для метода confirm. Это должно быть true или false:  $this->**confirm**($question, true); |

Регистрация команд

Как только ваша команда будет готова, нужно зарегистрировать её в Artisan-е, чтобы она была доступна для использования. Обычно это делается в файле app/Providers/ArtisanServiceProvider.php. В этом файле вы можете зарегистрировать команды в IoC контейнере и воспользоваться методом commands для их регистрации в Artisan-е. По умолчанию, образец регистрации команды включен в сервис-провайдер. Например:

|  |
| --- |
| **Регистрации команды. Листинг 1.** |
| $this->app->**bindShared**('commands.inspire', function()  {  return new InspireCommand;  }); |

Как только команда зарегистрирована в IoC контейнере, вы можете воспользоваться методом commands в вашем сервис-провайдере, чтобы сделать её доступной в Artisan-е. Вам необходимо передать название, использованное при регистрации команды в IoC контейнере:

|  |
| --- |
| **Делаем команду доступной в сервис-провайдере. Листинг 1.** |
| $this->commands('commands.inspire'); |

D

16. Service Providers

Основная задача поставщика услуг, или Service Providers – это загрузка вспомогательных, в том числе, и собственных классов.

Если открыть конфигурационный файл app.php, то мы увидим там множество провайдеров определенных в массиве providers. Это все классы, которые будут загружены для приложения. Многие из них являются отложенными, что означает, что они не будут загружаться при каждом запросе, а только тогда, когда они действительно необходимы.

Все провайдеры должны наследоваться от класса Illuminate\Support\ServiceProvider. Данный абстрактный класс требует, чтобы был определен хотя бы один метод класса-провайдера: register. В этом методе регистрируется вспомогательный класс.

Сам провайдер можно создать с помощью artisan:

|  |
| --- |
| **Создание провайдера с помощью artisan. Листинг 1.** |
| php artisan **make:provider** PhpqueryServiceProvider |

В папке providers/ появился файл PhpqueryServiceProvider.

Метод Register()

Откроем метод register и зарегистрируем новый класс:

|  |
| --- |
| **Регистрация нового класса провайдера. Листинг 1.** |
| public function register()  {  $this->app->**singleton**('phpQuery\phpQuery', function($app)  {  return new Connection($app['config']['riak']);  });  } |

В register() мы регистрируем (bind) как singleton, т.е. класс не будет переинициализироваться после вызова из контейнера.

Метод boot()

Пространство имени App\Providers, в котором находится этот класс сервис-провайдера - место для хранения сервис-провайдеров нашего Laravel-приложения, но мы можете располагать свои сервис-провайдеры где угодно внутри PSR-4 папки (если вы не меняли composer.json, то это папка app).

Когда вызвались методы register() всех сервис-провайдеров приложения, вызывается метод boot() сервис-провайдеров. Там уже можно использовать весь существующий функционал классов фреймворка и приложения - регистрировать слушателей событий, подключать роуты и т.п.

|  |
| --- |
| **Метод boot сервис-провайдеров. Листинг 1.** |
| public function boot(Dispatcher $events)  {  Event::listen('SomeEvent', 'SomeEventHandler');  } |

Создадим папку libs, в которой объявим класс

|  |
| --- |
| **Пользовательский класс для провайдера. Листинг 1.** |
| <?php **namespace App\Libs**;  class phpQuery {  public function foo(){  echo ‘test’;  }  } |

Регистрация провайдеров

Провайдеры регистрируются в файле config/app.php

|  |
| --- |
| **Регистрация провайдеров в файле confgig/app.php. Листинг 1.** |
| 'providers' => [  // другие сервис-провайдеры  'App\Providers\AppServiceProvider',  ], |