

Создание web-приложений,
исполняемых на стороне сервера при помощи
языка программирования PHP
и технологии AJAX

Урок №6

Laravel 5.3

Содержание

Введение	4
Особенности Laravel	5
Composer	6
Artisan	9
Описание Laravel приложения	10
Структура папок приложения	13
Конфигурирование приложения	14
Создание контроллера	16
Создание представления	21
Движок Blade	24

Новое приложение.	27
Создание приложения Larabook	28
Создание базового представления	30
Создание таблиц базы данных	33
Создание моделей	40
Подключение модуля Forms	43
Создание RESTful контроллера	46
Работа с формами	50
Валидация данных формы	57
Создание первой страницы	69
Расширение функционала	79
Аутентификация	83
Подведение итогов.	92

Введение

Нам предстоит ознакомиться еще с одним очень популярным PHP фреймворком – Laravel. Этот фреймворк, так же как и Codeigniter, работает по паттерну MVC. Это значит, что созданные в нем приложения будут состоять из контроллеров, моделей и представлений. Сразу хочу предупредить вас, что изучение Laravel – более серьезное испытание, чем изучение Codeigniter. Если вы помните, характеризуя Codeigniter мы говорили о том, что он очень прост в изучении и использовании? Сейчас вы сможете сравнить изучение Codeigniter и Laravel и убедитесь в том, что Codeigniter действительно очень простой фреймворк. Однако, вас не должны пугать трудности. Изучив Laravel, вы здорово повысите свою профессиональную значимость и пополните ряды веб – разработчиков, владеющих самыми современными инструментами.

Особенности Laravel

Процесс разработки приложения под управлением Codeigniter выглядит целостно и единообразно: вы пишете код и создаете разметку в разных файлах, расположенных в корневой папке вашего приложения. Вам достаточно текстового редактора для разработки и браузера с веб – сервером для тестирования приложения. С Laravel дела обстоят иначе. Для создания Laravel приложения вам понадобится еще ряд инструментов, среди которых есть довольно необычные. К тому же, приготовьтесь выполнять большой объем работы в консоли. Поначалу это отвлекает и может привести в замешательство, но когда вы поймете назначение каждого из этих инструментов, вы оцените их важность и эффективность. А затем привыкните к ним.

Перед установкой Laravel на вашем компьютере уже должны быть установлены:

- Apache;
- PHP;
- MySQL;
- Composer.

Вместо Apache, конечно же, можно использовать другой веб – сервер. PHP должен быть версии 5.4 или выше. Мы будем работать с версией PHP 5.6, чтобы использовать самую последнюю версию Laravel. С MySQL тоже все понятно. Вопросы может вызывать Composer – один из тех необычных инструментов, о которых мы только что говорили. Что же это такое?

Composer

Composer – это менеджер зависимостей для PHP. Большинство приложений сегодня зависят от различных внешних библиотек. Чтобы пользоваться такими библиотеками, их надо подключать в свое приложение. Часто таких библиотек бывает несколько, и тогда надо учитывать возможные проблемы совместимости между ними. Иногда еще надо знать, какие версии той или другой библиотеки следует использовать. Для загрузки библиотеки обычно скачивают архив и извлекают из него требуемые ресурсы в папки своего проекта.

Composer позволяет автоматизировать этот процесс. Вы можете указать, какие библиотеки необходимы для вашего проекта, а Composer определит, какие их версии требуются, и внедрит их в проект. Это очень полезный инструмент, используемый не только для Laravel, а для любых PHP проектов, зависящих от сторонних библиотек, которые, в свою очередь, тоже могут быть зависимости. Composer работает в CLI (command – line interface), т.е. это консольный инструмент.

К сказанному нужно добавить, что Composer – кроссплатформенный инструмент, работающий под управлением разных операционных систем. Установить этот пакет можно по ссылке <https://getcomposer.org>. На главной странице надо кликнуть по кнопке Download:

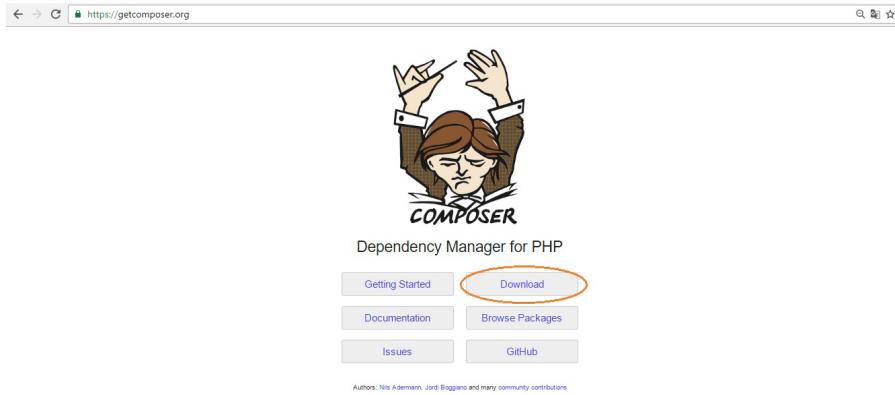


Рис. 1. Главная страница Composer

Затем на следующей странице надо активировать ссылку загрузки установщика:

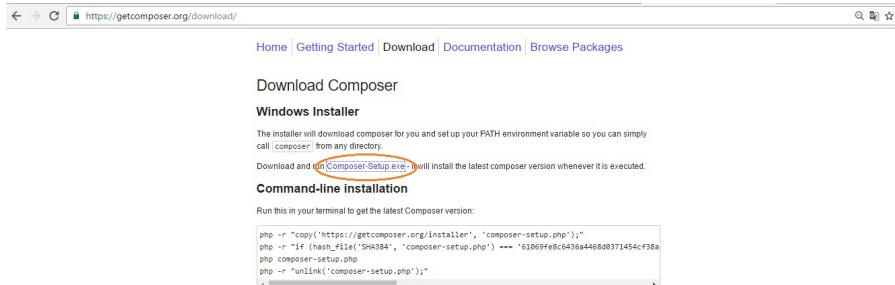


Рис. 2. Загрузка Composer

Таким образом, вы скачаете небольшой загрузчик, который выполнит установку Composer на ваш компьютер. Нам с вами этого делать не надо. Дело в том, что наш OpenServer включает в себя уже установленный Composer.

Теперь, когда все необходимые инструменты присутствуют на вашем компьютере, можно начинать со-

здание Laravel приложения. Необходимо лишь сделать небольшое отступление, чтобы ознакомиться с одним инструментом, без которого не обойтись при работе с Laravel.

Artisan

Artisan – еще одна утилита с консольным интерфейсом, выполняющая для разработчика большой объем рутинной работы. Она устанавливается вместе с Laravel, поэтому предпринимать какие – либо действия для ее установки не надо. Какие действия выполняет Artisan? Вот перечень основных:

- автоматически создает шаблоны кода для контроллеров, моделей и другого;
- создает, редактирует и удаляет объекты БД;
- выполняет начальное заполнение БД;
- управляет маршрутизацией;
- конфигурирует приложение;
- выполняет Unit тесты.

Чтобы увидеть список доступных команд artisan, надо в консольном окне, открытом в папке созданного приложения, выполнить такую команду:

```
php artisan list
```

Не пугайтесь большому количеству команд, которые вы увидите. Многие из них требуются редко. С самыми необходимыми мы познакомимся в ходе разработки приложения.

Описание Laravel приложения

При работе с Laravel нам постоянно надо будет использовать консольное окно. Вы должны помнить, что OpenServer включает в себя также собственное консольное окно. Выделите иконку OpenServer в трее и нажмите правую кнопку мыши. Затем активируйте пункты меню Advanced и Console. Запустится консольное окно. Не закрывайте его в ходе работы над приложением.

Прежде всего, запомните, что для создания нового приложения консольное окно должно быть открытым в корневой папке OpenServer. Поэтому выполните в нем команду:

```
cd "C:\OpenServer\domains\localhost"
```

Если ваш OpenServer установлен по другому пути, то, конечно же, укажите свой путь к папке localhost.

В дальнейшем, когда приложение будет создано и мы будем работать с ним, консольное окно по-прежнему надо будет держать открытым, но уже в корневой папке текущего приложения.

Теперь мы готовы создать наше Laravel приложение. В этом нам поможет Composer. Так как он является консольным инструментом, все команды для него мы будем вводить в консольном окне.

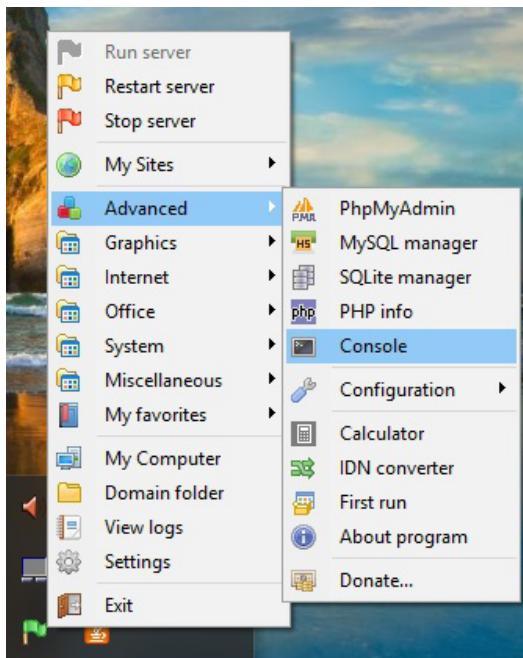


Рис. 3. Запуск консольного окна

Для создания нового Laravel приложения в консольном окне надо ввести такую команду:

```
composer create-project laravel/laravel laratest
--prefer-dist
```

В этой строке laratest – это имя создаваемого проекта. Приложение будет создано в папке с таким же именем, которая, в свою очередь, будет создана Composer'ом в папке localhost. Ключ --prefer-dist означает, что мы хотим создать приложение для той версии Laravel, которая подходит вашему окружению. (Обратите внимание, что перед словом prefer стоит два дефиса без пробела между ними.) О разных версиях Laravel поговорим позже.

Если вы хотите создать приложение для какой – то конкретной версии Laravel, то вместо этого ключа в данной команде надо указать желаемую версию. Например, так:

```
composer create-project laravel/laravel laratest "5.2.*"
```

Обратите внимание, что в ходе разработки приложения нам нужен доступ к Интернету. Если вы выходите в Интернет через прокси – сервер, то вам надо ввести в консольном окне такую команду:

```
set HTTP_PROXY=http://user:password@XX.XX.XX.XX:YYYY
```

где:

user – имя пользователя для прокси-сервера;

password – пароль для подключения к прокси;

XX.XX.XX.XX – адрес прокси-сервера;

YYYY – номер порта прокси-сервера.

Давайте создадим новое приложение с ключом --prefer-dist, для чего надо выполнить в консольном окне такую команду (убедившись еще раз, что консольное окно открыто в папке localhost):

```
composer create-project laravel/laravel laratest  
--prefer-dist
```

В ходе выполнения этой команды будут созданы все необходимые для приложения папки и загружены требуемые ресурсы. На момент написания этого урока текущая версия Laravel – v5.3.28.

Для проверки созданного приложения введите в браузере адрес <http://localhost/laratest/public/index.php>.

Если все сделано правильно, вы увидите стандартную страницу приветствия:



Рис. 4. Страница приветствия Laravel

Структура папок приложения

После выполнения последней команды Composer в папке localhost была создана папка с именем вашего приложения, в которой располагаются остальные папки приложения. Структура Laravel приложения более сложная, чем структура Codeigniter приложения. Будем знакомиться с папками и их содержимым постепенно.

app	28.12.2016 10:09
bootstrap	28.12.2016 10:09
config	28.12.2016 10:09
database	28.12.2016 10:09
public	28.12.2016 10:09
resources	28.12.2016 10:09
routes	28.12.2016 10:09
storage	28.12.2016 10:09
tests	28.12.2016 10:09
vendor	28.12.2016 10:10

Рис. 5. Структура папок приложения

Отметьте пока следующие папки:

- **public** – корневая папка приложения, здесь находится стартовая страница index.php и внешние ресурсы приложения: стили, скрипты, картинки;
- **app** – в этой папке будет находиться большинство кода, который мы будем писать (контроллеры и модели);
- **config** – в этой папке находятся конфигурационные файлы приложения;
- **resources** – в этой папке находятся представления приложения.

Об остальных папках будем говорить по мере необходимости. В некоторых из них мы вообще не будем ничего делать, в других иногда будем выполнять какие-либо действия.

Конфигурирование приложения

Как и в случае использования Codeigniter, созданное приложение надо настроить для конкретной работы. Настройка сводится к инициализации ряда конфигурационных массивов и выполняется в разных файлах в папке config.

В файле /config/app.php можно изменять такие настройки приложения, как:

- включение или отключение отладчика:

```
'debug' => env('APP_DEBUG', false)
```

- выбор часового пояса:

```
'timezone' => 'UTC'
```

- создание ключа для шифрования:

```
'key' => env('APP_KEY', 'SomeRandomString')
```

Для нашего текущего приложения здесь можно ничего не изменять.

В файле `/config/auth.php` указывается информация для аутентификации. Эту часть фреймворка мы рассмотрим позже.

В файле `/config/database.php` содержатся настройки для работы с БД. Поскольку мы планируем работать с MySQL, нам надо внести изменения в следующих строках этого файла:

```
'mysql' => [
    'driver'     => 'mysql',
    'host'        => env('DB_HOST', 'localhost:3307'),
    'database'   => env('DB_DATABASE', 'laratest'),
    'username'   => env('DB_USERNAME', 'root'),
    'password'   => env('DB_PASSWORD', '123456'),
    'charset'    => 'utf8',
    'collation'  => 'utf8_unicode_ci',
    'prefix'     => '',
    'strict'     => false,
],
```

Обратите внимание на номер порта у адреса СУБД – его надо указывать явно.

Заметьте также, что в PHP, начиная с версии PHP 5.4, массив можно описывать с помощью прямоугольных скобок, как указано в этом конфигурационном файле. Другими словами, обе следующие строки создают одинаковые массивы:

```
$ar = array(1, 2, 3, 4, 5);  
$ar1 = [1, 2, 3, 4, 5];
```

Кроме этого, надо внести некоторые изменения в файл .env, расположенный в корне папки laratest. Откройте этот файл в блокноте и приведите строки 8, 10, 11 и 12 к такому виду:

```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1:3307  
DB_PORT=3306  
DB_DATABASE=laratest  
DB_USERNAME=root  
DB_PASSWORD=123456
```

Отметьте, что кавычки мы здесь не используем. Такой начальной настройки достаточно, чтобы двигаться дальше.

Создание контроллера

Laravel – это фреймворк, использующий MVC архитектуру. Вспомним, что мы знаем о паттерне MVC. В паттерне MVC контроллер – это класс, с которым непосредственно работает пользователь приложения. Работа с этим классом сводится к тому, что пользователь вызывает тот или другой метод контроллера. Другими словами, каждый метод контроллера реализует какое – либо действие, которое пользователь может выполнить в приложении. Методы контроллера так и называются – действия.

Большинство действий пользователя требуют обращения к БД. По законам паттерна MVC контроллер

не имеет права доступа к БД. За работу с ней отвечает класс модели. Так вот, если пользователь вызвал какой – либо метод контроллера и для выполнения действия требуется обращение к БД, то контроллер из своего метода вызовет необходимый метод модели. Метод модели напрямую обратится к БД и выполнит требуемый в данной ситуации SQL запрос. Если запрос возвращает какое – либо значение, то оно будет возвращено, конечно же, в метод контроллера, вызвавший метод модели. Метод контроллера проверит полученные из модели данные и, в зависимости от результатов проверки, выведет пользователю на экран то или другое представление. Например, страницу, отображающую список товаров из БД, или страницу, сообщающую об отсутствии товаров или об ошибке выполнения запроса к БД.

Зафиксируйте на данный момент такой сценарий: контроллер обращается к модели, получает от модели результат работы с БД, а затем, в зависимости от полученного результата, выводит пользователю какую – либо html страницу (представление).

Также возможен вариант использования контроллера без модели. Именно так и будет в первом варианте нашего приложения. Таким образом, контроллер является центральным звеном в работе пользователя с приложением. Laravel позволяет нам создавать разные контроллеры. Мы можем создать пустой класс контроллера и добавлять в него требуемые для нашего приложения действия (методы). А можем создать так называемый RESTful контроллер с уже готовыми заготовками действий. О RESTful контроллере мы поговорим подробнее

позже, когда добавим в наше приложение модель. Сейчас рассмотрим создание пустого контроллера.

Как вы помните, контроллеры создаются с помощью консольной утилиты artisan.

В консольном окне, открытом в корневой папке созданного приложения надо выполнить команду:

```
php artisan make:controller TestController
```

Если все прошло успешно, вы увидите в консольном окне сообщение **Controller created successfully**.

У меня это выглядит так:

The screenshot shows a Windows Command Prompt window titled "cmd". The command line shows the user navigating to the application directory and running the artisan command to create a controller. Two specific lines of the command history are highlighted with yellow ovals:

- The command "cd C:\OpenServer\domains\localhost\laratest" is highlighted with a yellow oval labeled "move to the app directory".
- The command "php artisan make:controller TestController" is highlighted with a yellow oval labeled "create controller".

```
cmd
Microsoft Windows [Version 10.0.14393]
admin@IDEA-PC ~ > cd C:\OpenServer\domains\localhost\laratest
admin@IDEA-PC C:\OpenServer\domains\localhost\laratest > php artisan make:controller TestController
Controller created successfully.

admin@IDEA-PC C:\OpenServer\domains\localhost\laratest >
```

Рис. 6. Создание контроллера

Контроллер для нас создала утилита artisan. Как вы поняли, TestController – это имя создаваемого контроллера. Договоримся, что все контроллеры будут называться по такому шаблону: XXXController. После выполнения этой команды в папке \laratest\app\Http\Controllers вы обнаружите файл TestController.php с таким содержимым:

```
<?php  
namespace App\Http\Controllers;  
use Illuminate\Http\Request;  
  
class TestController extends Controller  
{  
    //  
}
```

Это и есть заготовка класса контроллера. Знакомая картина, не так ли? Мы снова видим, что созданный нами класс контроллера является производным от системного класса Controller. Следовательно, наш TestController содержит в себе функционал своего базового класса.

Как видите, в этом случае нет никаких методов. Методы мы должны создавать самостоятельно, в зависимости от наших потребностей. Посмотрим, как пользователь приложения может работать с методами контроллера. Для вызова каждого метода мы должны добавить в приложение адрес, который будет соответствовать вызову именно этого метода. Адресация приложения описывается в файле \laratest\routes\web.php. Откройте этот файл. К имеющемуся там коду добавьте такую строку:

```
Route::get('/home/', 'TestController@index');
```

Теперь содержимое файла выглядит так:

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

Добавленная строка означает, что при вводе пользователем в адресной строке адреса <http://localhost/laratest/public/index.php/home> будет вызван метод index() контроллера TestController. Обратите внимание, вы не указываете в адресной строке имя контроллера и имя метода. Вместо этого вы указываете имя маршрута, ассоциированного с контроллером и методом. В нашем случае мы придумали имя маршрута home.

Добавьте в класс нашего контроллера метод index():

```
class TestController extends Controller
{
    function index()
    {
        return 'From index method';
    }
}
```

Сохраните все изменения и введите в браузере адрес: <http://localhost/laratest/public/index.php/home>.

Если вы все сделали правильно, то увидите на экране строку, возвращенную из метода index():

```
From index method
```

Этот пример демонстрирует работу с контроллером и применение маршрутизации в файле `\laratest\routes\web.php`. Обратите внимание, что в нашем приложении мы пока не использовали никаких моделей и никакой БД.

Подведем промежуточный итог. Заготовка контроллера создается с помощью `artisan`. Мы создали пустой контроллер, хотя существуют и другие возможности. Имя класса контроллера должно включать в себя слово `Controller`. Класс контроллера располагается в папке `\laratest\app\Http\Controllers` в файле, имя которого совпадает с именем класса. Для вызова метода контроллера надо прописать маршрут в файле `\laratest\routes\web.php`. Скоро мы научимся одной строкой в этом файле прописывать маршруты сразу для нескольких методов.

Создание представления

Давайте теперь рассмотрим, как приложение может возвращать пользователю в ответ на введенный адрес не просто строку из метода контроллера, а какое-нибудь представление.

Представления располагаются в папке `\laratest\resources\views` и представляют собой файлы с двойным расширением `.blade.php`. Blade – это движок разметки или шаблонизатор, используемый Laravel для размещения кода внутри html разметки. Для чего нужен код в представлении? Ведь паттерн MVC не приветствует вынесение пользовательской логики в представление. Другими словами, представление должно только отображать полученные из контроллера данные, но не вы-

полнять их обработку. Однако для отображения данных часто тоже надо выполнять некоторую обработку. Вот для этого и существует Blade. Сейчас вы увидите, как он работает.

Представления удобно группировать по контроллерам и все представления одного контроллера хранить в одной папке, вложенной в папку `\laratest\resources\views`. Давайте создадим папку `test` по пути `\laratest\resources\views`. Внутри созданной папки `test` создадим файл с именем `index.blade.php`. Приведите содержимое созданного файла к такому виду:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Laravel</title>
    </head>
    <body>
        <div class="container">
            <div class="content">
                <div class="title">Laravel 5 from
index()</div>
            </div>
        </div>
    </body>
</html>
```

Теперь давайте вызовем это представление из нашего метода `index()`. Для этого надо привести метод `index()` к такому виду:

```
function index()
{
    return view('test.index');
}
```

Сохраните все изменения и снова введите в браузере адрес: <http://localhost/laratest/public/index.php/home>.

Если вы все сделали правильно, то увидите на экране строку, возвращенную из представления:

```
index.blade.php: Laravel 5 from index()
```

Обратите внимание на то, как мы указали имя представления в методе `view()`, вызванном в методе `index()`. Во-первых, мы не указывали расширение `.blade.php`, т.к. фреймворк сам понимает, о чём идет речь. Во-вторых, перед именем представления мы указали `test.` – это признак того, что наше представление вложено в папку с именем `test`. Другими словами, чтобы вывести какое-либо представление, надо в вызванном методе контроллера вызвать метод `view()` и передать в него в виде строки путь к требуемому представлению без расширения, но с учетом вложенности. При этом в качестве разделителя между именем папки и именем файла представления используется символ точки `«.»`.

Например, если бы наше представление лежало непосредственно в папке `\laratest\resources\views`, а не в какой-то вложенной папке, то строка вызова этого представления выглядела бы так:

```
function index()
{
    return view('index');
}
```

Как видите, здесь не указана вложенная папка `test`.

Движок Blade

В предыдущем разделе мы кратко охарактеризовали шаблонизатор Blade и рассказали для чего он используется и что умеет делать. Однако в представлении index.blade.php Blade не использовался. Сейчас мы выполним пример, в котором рассмотрим некоторые аспекты его применения. Кстати, возможно, вам интересно будет узнать, что разработчики Blade взяли за основу синтаксис движка Razor из ASP.NET. Присмотритесь внимательнее, и вы заметите сходство.

Мы с вами использовали самый простой способ вызова метода view(), при котором передали в метод только путь к требуемому представлению. Однако, мы можем при вызове этого метода не только указать, какое представление отобразить, а также еще и передать в это представление какие – либо данные. Для этого надо вызывать метод view() с двумя параметрами: первый – это строка с путем к представлению, а второй —ассоциативный массив с передаваемыми значениями. Со значениями, переданными в представление, можно выполнять разные действия. Их можно просто выводить на странице, можно выполнять сравнения, циклическую обработку и т.п. Все эти действия реализует Blade.

Рассмотрим, как это работает. Для этого изменим вызывающий метод index():

```
function index()
{
    $r = rand(0,1000);           //just a random value
                                //from 0 to 999
    $countries =
    array("Argentina", "Belgiun", "Canada", "Denmark");
    //an array
```

```
    return view('test.  
index',array('p1'=>$r,'p2'=>$countries));  
}
```

Вы уже видели подобный прием, когда передаваемые в представление данные упаковываются в массив. Такой вызов передает в представление два параметра с именами p1 и p2. В нашем случае p1 – это просто случайное число, а p2 – массив с названиями стран. Посмотрим, как с этими значениями можно работать в представлении:

```
<body>  
    <div class="container">  
        <div class="content">  
            <div class="title">Laravel 5 from  
                index()</div>  
            <p> Just output of random value:  
                {{ $p1 }}</p>  
            @if ($p1 > 700)  
                <p>greater</p>  
            @else  
                <p>less</p>  
            @endif  
            <ul>  
                @foreach ($p2 as $p)  
                    <li>  
                        {{ $p }}  
                    </li>  
                @endforeach  
            </ul>  
        </div>  
    </div>  
</body>
```

Снова активируйте в браузере адрес <http://localhost/laratest/public/index.php/home> и посмотрите на результат. У меня он выглядит так:

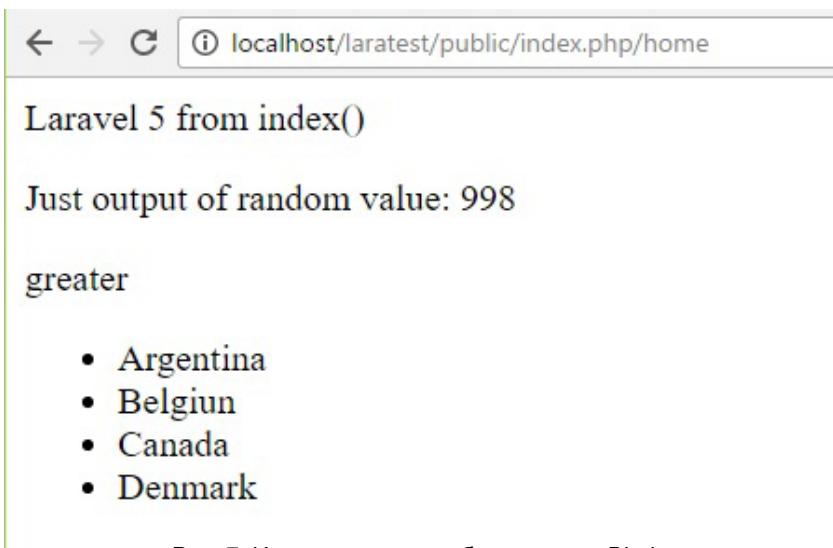


Рис. 7. Использование шаблонизатора Blade

В этом представлении вы увидели использование Blade. Запомните, что такие скобки {{ }} означают выражение. Все, что расположено внутри этих скобок, вычисляется, и полученное значение передается в браузер. Вы также должны запомнить, что символ «@» является признаком кода движка Blade.

Особого смысла в этих действиях нет, но они демонстрируют часть того, что умеет делать Blade. В данном примере есть и отображение значения переменной, и выполнение условной обработки, и циклическая обработка массива. С другими возможностями Blade будем знакомиться по ходу разработки нашего нового приложения.

Новое приложение

Предыдущей информации о Laravel достаточно для того, чтобы начать создание полноценного приложения с моделями и базой данных. При разработке этого приложения мы рассмотрим новые возможности фреймворка и подробнее разберемся с уже рассмотренными.

Назовем наше приложение larabook. Это приложение будет представлять собой электронную книгу. Книга будет состоять из разделов (Topic), а каждый такой раздел будет состоять из набора блоков (Block). Каждый же блок, в свою очередь, будет содержать текстовый или графический контент.

Мы создадим в нашем приложении две страницы. Первая будет отображать в левой колонке список разделов. При клике по какому-либо разделу в центральной части страницы будут отображаться его блоки. Вторая страница будет содержать форму для добавления новых разделов и блоков.

Выполняя это задание, мы научимся работать с БД, создавать формы, выполнять валидацию данных форм и отображать в представлении результат выполненных запросов.

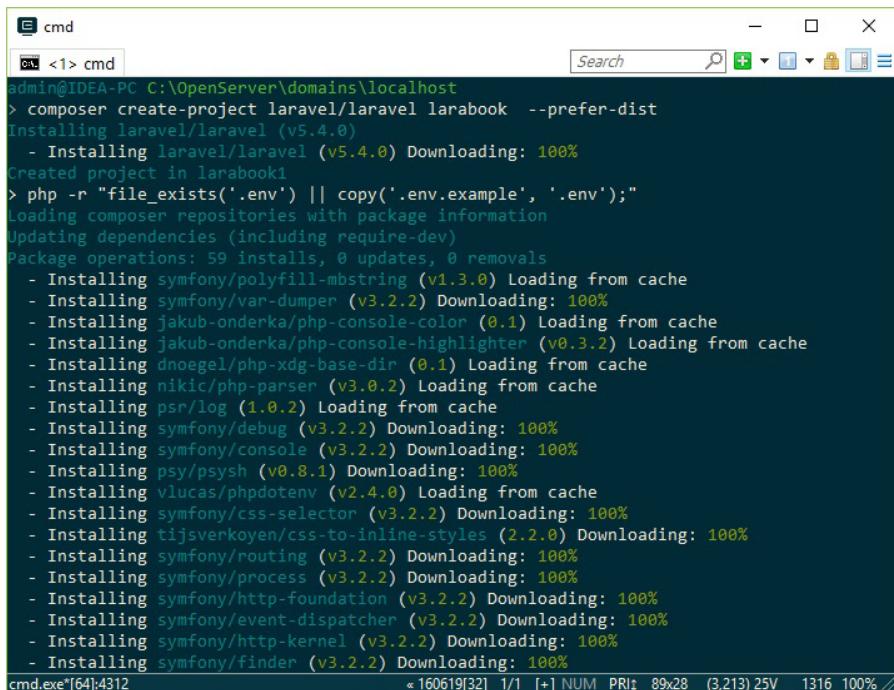
Для каждой из этих страниц нам надо будет создать свое представление. Кроме этих двух представлений, мы создадим еще и другие, и среди них – главное представление нашего приложения. Каждая страница приложения будет производной от этого главного представления, которое будет отображать общий для всех страниц контент.

Создание приложения Larabook

Создадим новое приложение с ключом – – prefer – dist, для чего надо выполнить в консольном окне такую команду (убедившись еще раз, что окно открыто в папке localhost):

```
composer create-project laravel/laravel larabook
--prefer-dist
```

Выполнение этой команды может занять несколько минут, поскольку в ходе создания приложения будет загружаться много различных библиотек и модулей Laravel. У меня создание приложения выглядит так:



```
cmd <1> cmd
admin@IDEA-PC C:\OpenServer\domains\localhost
> composer create-project laravel/laravel larabook --prefer-dist
Installing laravel/laravel (v5.4.0)
- Installing laravel/laravel (v5.4.0) Downloading: 100%
Created project in larabook1
> php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 59 installs, 0 updates, 0 removals
- Installing symfony/polyfill-mbstring (v1.3.0) Loading from cache
- Installing symfony/var-dumper (v3.2.2) Downloading: 100%
- Installing jakub-onderka/php-console-color (0.1) Loading from cache
- Installing jakub-onderka/php-console-highlighter (v0.3.2) Loading from cache
- Installing dnoegel/php-xdg-base-dir (0.1) Loading from cache
- Installing nikic/php-parser (v3.0.2) Loading from cache
- Installing psr/log (1.0.2) Loading from cache
- Installing symfony/debug (v3.2.2) Downloading: 100%
- Installing symfony/console (v3.2.2) Downloading: 100%
- Installing psy/psysh (v0.8.1) Downloading: 100%
- Installing vlucas/phpdotenv (v2.4.0) Loading from cache
- Installing symfony/css-selector (v3.2.2) Downloading: 100%
- Installing tijssverkojen/css-to-inline-styles (2.2.0) Downloading: 100%
- Installing symfony/routing (v3.2.2) Downloading: 100%
- Installing symfony/process (v3.2.2) Downloading: 100%
- Installing symfony/http-foundation (v3.2.2) Downloading: 100%
- Installing symfony/event-dispatcher (v3.2.2) Downloading: 100%
- Installing symfony/http-kernel (v3.2.2) Downloading: 100%
- Installing symfony/finder (v3.2.2) Downloading: 100%
```

Рис. 8. Создание нового приложения

Проверьте, создалась ли в папке localhost вложенная папка нового приложения с именем larabook. Если все в порядке, выполните начальную настройку нового приложения, как мы это делали в предыдущей части текущего урока.

В файле `/config/database.php` выполните настройки для работы с СУБД MySQL и с базой данных по имени larabook:

```
'mysql' => [
    'driver'     => 'mysql',
    'host'        => env('DB_HOST', 'localhost:3307'),
    'database'   => env('DB_DATABASE', 'larabook'),
    'username'   => env('DB_USERNAME', 'root'),
    'password'   => env('DB_PASSWORD', '123456'),
    'charset'    => 'utf8',
    'collation'  => 'utf8_unicode_ci',
    'prefix'     => '',
    'strict'     => false,
],
```

Кроме этого, надо внести изменения в файл `.env`, расположенный в корне папки larabook. Откройте этот файл в блокноте и приведите строки 8, 10, 11 и 12 к такому виду:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1:3307
DB_PORT=3306
DB_DATABASE=larabook
DB_USERNAME=root
DB_PASSWORD=123456
```

Снова отметьте, что кавычки мы здесь не используем. Настройка нового приложения выполнена.

В нашем приложении будет два контроллера: TopicController и BlockController. У каждого из них будет своя модель. У каждой модели будет соответствующая таблица в БД. Для каждой модели и ее таблицы объектно – реляционное соответствие (ORM) будет создаваться средствами Laravel. В состав Laravel входит специальная компонента Eloquent ORM, предназначенная для реализации соответствия между классами моделей и таблицами БД. И, конечно же, у каждого контроллера будут свои представления.

Мы уже говорили, что для удобства работы с приложением полезно хранить представления контроллеров в разных папках. Каждое представление вызывается из какого-либо контроллера. Поэтому удобно в папке \larabook\resources\views создать свою папку для каждого контроллера и в них хранить соответствующие представления. У нас будет два контроллера, и представления каждого контроллера будем хранить в отдельной вложенной папке. Нам надо создать по пути \larabook\resources\views две папки – topic и block. Папки называйте в нижнем регистре. Кроме этого, рядом с этими двумя папками еще надо создать папку layouts. В ней мы создадим базовое представление для нашего приложения, на которое будут ссылаться представления контроллеров.

Создание базового представления

Создадим одно представление, которое будет базовым для всех страниц нашего приложения. В нем мы создадим заголовок, меню и будем все это отображать

на остальных страницах. Все страницы нашего приложения будут наследовать это базовое представление. Для базового представления мы создали папку по пути \larabook\resources\views\layouts и в ней теперь создадим файл с именем master.blade.php.

Конечно же, мы будем использовать в нашем приложении Bootstrap. Чтобы применять его для наших страниц, скопируйте в папки css и js в папке \laratest\public bootstrap'овские файлы. Там же создайте папку images. Приведите содержимое файла master.blade.php к такому виду:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf - 8">
        <meta name="viewport" content="width=device -
            width, initial - scale=1.0">
        <meta name="description" content="">
        <meta name="author" content="">
        <title>Larabook</title>
        <link href="{{asset('css/style.css')}}">
            rel="stylesheet"
        <link href="{{asset('css/bootstrap.min.
            css')}}"> rel="stylesheet">
        <link href="{{asset('css/bootstrap - theme.
            min.css')}}"> rel="stylesheet">
        <!-- Latest compiled and minified
            JavaScript -->
        <script type="text/javascript"
            src="{{asset('js/bootstrap.min.js')}}">
            </script>
        <script type="text/javascript"
            src="{{asset('js/jquery - 3.0.0.js')}}">
            </script>

    </head><!-- /head -->
```

```

<body>
    @section('menu')
    <div class="mainmenu1 col - sm - 12 col - md
              - 12 col - lg - 12">
        <ul class="nav nav - pills nav -
                  justified">
            <li role="presentation" {{$page ==
                'Main page' ? 'class=active' : ''}}>
                <a href="{{url('index')}}">Main
                    Page</a></li>
            <li role="presentation" {{$page ==
                'Forms' ? 'class=active' : ''}}>
                <a href="{{url('create')}}">
                    Content Control</a></li>
        </ul>
    </div>
    @show

    <div class="container col - sm - 12 col - md
              - 12 col - lg - 12">
        @yield('content')
    </div>
</body>
</html>

```

Обратите внимание, каким образом выполняется ссылка на подключаемые файлы в тегах script и link – с использованием выражений Blade. Выражения Blade также используются в элементах li, где вычислением тернарного оператора определяется, добавлять или не добавлять элементу атрибут 'class=active'. Кроме этого, вы видите еще несколько директив Blade: @section, @show и @yield. Перечислим назначение этих и некоторых других директив:

- @section – открывает именованную секцию;

- `@show` – закрывает именованную секцию, открытую директивой `@section`, и сразу же выводит содержимое этой секции;
- `@endsection` – закрывает именованную секцию, без возможности ее дополнения;
- `@append` – закрывает секцию и добавляет ее содержимое к существующей секции с таким же именем;
- `@overwrite` – закрывает секцию, перезаписывая содержимое секции с таким же именем;
- `@yield` – выводит контент именованной секции по имени.

Мы создали главную страницу с двумя пунктами меню: `Main Page` и `Content Control`. Теперь, когда мы будем создавать другие представления, мы будем ссылаться из них на эту страницу.

Создание таблиц базы данных

При настройке нашего нового приложения мы указали имя БД `larabook`. Пока еще такой БД у нас нет, но уже пора создать ее. Запустите `phpmyadmin` и создайте базу данных с таким именем.

Теперь мы создадим в нашей БД несколько связанных таблиц. Обратите особое внимание на то, как Laravel создает таблицы в БД. При работе с БД Laravel использует миграции. Это понятие должно быть вам знакомо еще со времен изучения Entity Framework. Миграция – это действия с БД, позволяющие как выполнять действия, изменяющие БД, так и отменять эти изменения, возвращая состояние БД к предыдущему состоянию. Кроме того, при работе с базой данных Laravel автома-

тически поддерживает ORM, понимая, что за каждой таблицей в БД существует класс модели в приложении.

Запомните: в среде Laravel разработчиков, принято называть таблицы во множественном числе, а модели – в единственном. Это правило является особенно почитаемым, и нарушать его не стоит. Поэтому наши таблицы будут называться Topics и Blocks.

Создадим в нашей БД две связанные таблицы:

Topics (id, topicname);

Blocks (id, topicid, title, content, imagePath).

В таблице Blocks поле topicid будет внешним ключом для связи с таблицей Topics. Поле title будет содержать заголовок блока, поле content – текстовое содержимое блока, а поле imagePath – путь к графическому контенту. Блок может не содержать текстовый и графический контент. Другими словами, поля content и imagePath могут содержать значение NULL.

Для создания таблиц в БД нам снова понадобится artisan. В консольном окне, открытом в корневой папке приложения, выполним команду:

```
php artisan make:migration create_tables
```

Если вы все сделали правильно, то после выполнения увидите в консольном окне такое сообщение:

```
Created Migration: 2017_01_01_131029_create_tables
```

Эта команда создала в папке \larabook\database\migrations файл с именем 2017_01_01_131029_create_tables.php. Обратите внимание, что в имя файла вклю-

чена метка с датой и временем. Дело в том, что файлов миграции в приложении бывает много, и по этой метке Laravel определяет, в какой очередности выполнять миграции. В этом файле мы пропишем создание необходимых нам таблиц. Изначально содержимое файла может быть таким:

```
<?php

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

class CreateTables extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

В созданном классе CreateTables содержатся два метода – up() и down(). Этот класс используется при выполнении миграции – действия, изменяющего состояние БД. При выполнении апгрейда работает метод up(), в котором, как правило, описывается создание новых объектов базы данных. При выполнении отката или даунгрейда работает метод down(), в котором, как правило, удаляется то, что создавалось в методе up(). Создание и изменение БД выполняется как цепочка миграций, которая позволяет выполнять изменения в обоих направлениях. Пропишем в этом файле в методе up() создание требуемых таблиц, в методе down() – их удаление:

```
<?php

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

class CreateTables extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('Topics', function(Blueprint
$table)
        {
            $table - >increments('id');
            $table - >string('topicname',100) ->unique();
            $table - >timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('Topics');
    }
}
```

```

Schema::create('Blocks', function(Blueprint $table)
{
    $table->increments('id');
    $table->integer('topicid')-
        >unsigned();
    $table->foreign('topicid')-
        >references('id')->on('Topics')-
            >onDelete('cascade');
    $table->string('title',100);
    $table->longText('content')-
        >nullable();
    $table->string('imagesPath',255)-
        >nullable();
    $table->timestamps();
});
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::drop('Topics');
    Schema::drop('Blocks');
}
}

```

Рассмотрим написанный код метода up(). В этом методе мы дважды вызываем статический метод create() класса Schema. Этот метод создает таблицы в той БД, к которой подключено приложение. Первым параметром мы передаем этому методу имя таблицы, которую хотим создать. Второй параметр – это анонимная функция, в которой надо описать поля создаваемой таблицы и их характеристики. Тип параметра этой функции –

некий класс Bluebird. Вы догадались, что классы Schema и Bluebird описаны где – то в базовом классе Migration. Методы класса Bluebird очень красноречиво говорят о том, что они выполняют.

- **\$table – >increments('id')** – создает автоинкрементное поле с именем 'id', которое будет первичным ключом в таблице;
- **\$table – >string('topicname',100) – >unique()** – создает строковое поле с именем 'topicname' длиной в 100 символов и накладывает ограничение уникальности на значения этого поля;
- **\$table – >timestamps()** – создает в текущей таблице два поля типа timestamp с именами 'created_at' и 'updated_at'; эти поля обслуживаются фреймворком самостоятельно, и в них заносятся дата создания строки таблицы и ее последнего изменения, соответственно. Создавать эти поля не обязательно, но их присутствие оказывается полезным во многих случаях;
- **\$table – >integer('topicid') – >unsigned()** – создает поле с именем 'topicid' типа unsigned integer;
- **\$table – >foreign('topicid') – >references('id') – >on('Topics') – >onDelete('cascade')** – создает связь между таблицами Blocks и Topics;
- **\$table – >longText('content')** – создает поле с именем 'content' типа longText;
- **\$table – >string('imagesPath',255)** – создает строковое поле с именем 'imagesPath' длиной в 255 символов.

Отметьте для себя, что в больших проектах правильнее будет создавать отдельный файл миграции для

каждой таблицы. Это упростит работу с БД при необходимости выполнять откаты и изменения.

Если вы занесли требуемый код в методы `up()` и `down()`, то теперь можете выполнить миграцию, которая создаст в нашей БД две таблицы, `Blocks` и `Topics`, согласно указанным в методах `Schema::create()` спецификациям. Перейдите в консольное окно, убедитесь, что оно открыто в корневой папке приложения, и выполните такую команду:

```
php artisan migrate
```

Если надо отменить последние сделанные в БД изменения, выполните команду:

```
php artisan migrate:rollback
```

Если же вы хотите откатить все сделанные в БД изменения, а потом создать БД сначала, выполните команду:

```
php artisan migrate:refresh
```

Итак, выполните команду:

```
php artisan migrate
```

а затем перейдите в `phpmyadmin` и убедитесь, что наши таблицы созданы. Кроме них вы увидите другие таблицы, но о них мы поговорим позже.

Создание моделей

Теперь у нашего приложения есть собственная БД. Давайте рассмотрим, как Laravel позволяет добавлять значения в БД, читать информацию из нее и выводить информацию пользователю в представления. Вот здесь возникает необходимость в моделях.

С точки зрения MVC паттерна, модель – это класс, содержащий в себе логику работы с БД. Но, кроме этого, модель полезно рассматривать как сущность, с которой имеет дело ваше приложение. Когда вы проектируете для своего приложения БД, вы, как правило, выясняете, с какими сущностями надо работать в приложении, и создаете для таких сущностей таблицы в БД.

Почти всегда бывает полезно создавать модели приложения так, чтобы они соответствовали таблицам в БД. Вы уже должны помнить, что этим соответствием занимаются ORM системы. Мы говорили, что Laravel имеет собственную ORM систему с именем Eloquent ORM. Именно Eloquent ORM будет выполнять в нашем приложении сопоставление классов моделей с таблицами БД, передавать в БД SQL запросы нашего приложения и получать обратно из БД результаты их выполнения. Работая над нашим приложением, мы научимся работать с Eloquent ORM.

В приложении будет правильно создать две модели, Topic и Block, и ассоциировать их с таблицами Topics и Blocks соответственно.

В Codeigniter модель представляет собой контейнер для методов, работающих с базой данных. Поэтому, создавая там класс модели, мы уделяли основное внимание

разработке методов в этом классе. В Laravel ситуация немного другая. Мы вообще не будем создавать в классе модели методы для работы с БД. Дело в том, что эти методы уже созданы разработчиками Laravel в базовом классе Model. Сейчас модель для нас – это прежде всего структурная единица хранения данных. Например, объект модели topic – это описание раздела нашей книги, объект модели block – набор данных одного блока.

Для создания модели Topic, соответствующей таблице Topics, надо выполнить такую artisan команду:

```
php artisan make:model Topic
```

При успешном выполнении команды вы получите в консольном окне сообщение Model created successfully, а в папке /app/ будет создан файл Topic.php. Откройте его в блокноте и приведите к такому виду:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Topic extends Model
{
    protected $primaryKey='id';
    protected $table='topics';
    protected $fillable=['topicname', 'created_at',
                        'updated_at'];
}
```

Вы видите определение класса Topic, наследующего классу Model. В этом классе определены некоторые свойства:

- `$primaryKey='id'` – указывает, какое поле в таблице будет первичным ключом;
- `$table='topics'` – указывает, что соответствует этому классу модели таблица 'topics';
- `$fillable=['topicname','created_at','updated_at']` – указывает, значения каких полей можно будет изменять в дальнейшем в коде.

Для создания модели Block, соответствующей таблице Blocks, надо выполнить такую artisan команду:

```
php artisan make:model Block
```

При успешном выполнении команды вы получите в консольном окне сообщение **Model created successfully**, а в папке /app/ будет создан файл Block.php. Откройте его в блокноте и приведите к такому виду:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Block extends Model
{
    protected $primaryKey='id';
    protected $table='blocks';
    protected
        $fillable=array('title','topicid','content',
                        'imagesPath',
                        'created_at','updated_at');
}
```

Вы уже понимаете, что означают указанные в классе этой модели свойства. Теперь в нашем приложении

созданы две модели. Рассмотрим, как их можно использовать.

Подключение модуля Forms

Как вы понимаете, нам придется много работать с формами. В Codeigniter для работы с ними существует хелпер Forms. В Laravel для этого существует специальный модуль под названием Forms. По умолчанию в Laravel 5 этот модуль не установлен. Поэтому сейчас рассмотрим установку модуля Forms.

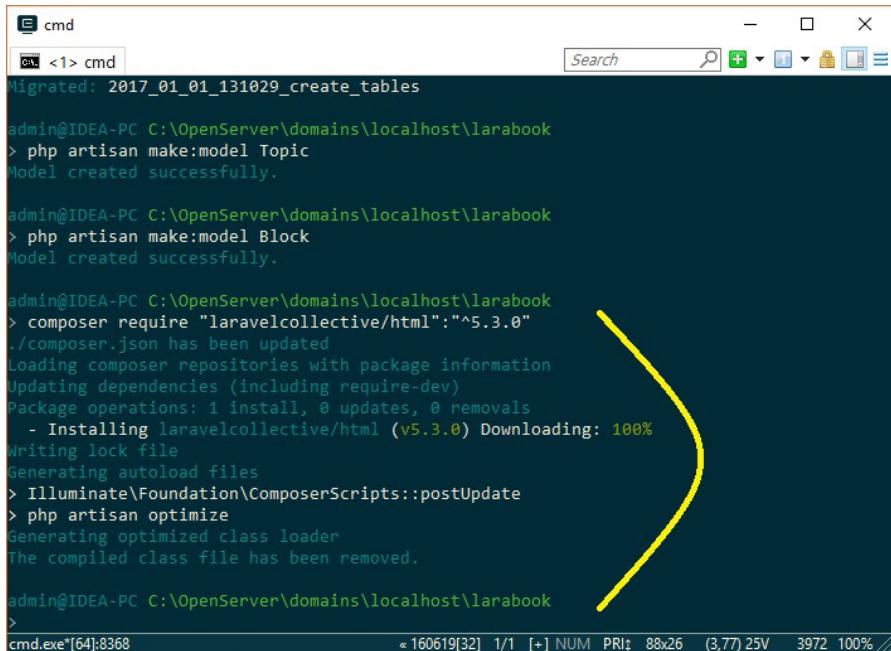
В корневой папке приложения есть файл composer.json. Этот файл управляет конфигурацией Laravel приложения. Если вам надо добавить в состав приложения какой-либо модуль, не установленный по умолчанию, вы можете сделать это в файле composer.json.

Откройте этот файл в блокноте. Вы увидите в нем JSON описание довольно сложного объекта. Добавляя и изменяя свойства этого объекта, мы можем управлять конфигурацией Laravel в нашем приложении. Ранее любые изменения надо было вручную прописывать в этом файле, после чего специальной командой файл надо было обновлять. Сейчас это уже делать не надо. Подобные изменения будет выполнять composer. Поэтому закройте этот файл без каких – либо изменений. Я попросил вас заглянуть в него, чтобы вы, с одной стороны, понимали его назначение, и с другой – еще раз оценили полезность composer.

Для установки модуля Forms в нашем приложении введите в консольное окно, открытое в корневой папке приложения, такую команду:

```
composer require "laravelcollective/html":'^5.3.0'
```

Выполнение этой команды может занять несколько минут, в зависимости от качества вашего соединения с Интернетом. При успешном завершении установки вы увидите такой вывод в консольном окне:



```
cmd
<1> cmd
Migrated: 2017_01_01_131029_create_tables
admin@IDEA-PC C:\OpenServer\domains\localhost\larabook
> php artisan make:model Topic
Model created successfully.

admin@IDEA-PC C:\OpenServer\domains\localhost\larabook
> php artisan make:model Block
Model created successfully.

admin@IDEA-PC C:\OpenServer\domains\localhost\larabook
> composer require "laravelcollective/html":'^5.3.0"
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing laravelcollective/html (v5.3.0) Downloading: 100%
Writing lock file
Generating autoload files
> Illuminate\Foundation\ComposerScripts::postUpdate
> php artisan optimize
Generating optimized class loader
The compiled class file has been removed.

admin@IDEA-PC C:\OpenServer\domains\localhost\larabook
>
cmd.exe[64]:8368 160619[32] 1/1 [+] NUM PRI: 88x26 (3,77) 25V 3972 100% //
```

Рис. 9. Установка модуля Forms

Теперь модуль Forms включен в состав Laravel нашего приложения, но для того, чтобы пользоваться им, надо сделать еще пару настроек в файле /config/app.php. Откройте этот файл в блокноте, найдите в нем описание элемента массива с именем 'providers' (строка 138), который сам является массивом, и добавьте к его значениям такое:

```
Collective\Html\HtmlServiceProvider::class,
```

Затем найдите элемент 'aliases' (строка 195) и к его значениям добавьте два таких:

```
'Form' => Collective\Html\FormFacade::class,
'Html' => Collective\Html\HtmlFacade::class,
```

Обратите внимание на такую важную деталь. Вы, скорее всего, и в массив 'providers' и в массив 'aliases' добавили новые строки в конец, после последнего существовавшего там значения. Так вот, даже если добавленные вами строки являются последними в массиве, обязательно оставьте после них символ запятой «,». Хотя добавить эти строки можно было в любое место.

```

160      Illuminate\Auth\Passwords>PasswordResetServiceProvider::class,
161      Illuminate\Session\SessionServiceProvider::class,
162      Illuminate\Translation\TranslationServiceProvider::class,
163      Illuminate\Validation\ValidationServiceProvider::class,
164      Illuminate\View\ViewServiceProvider::class,
165      Collective\Html\HtmlServiceProvider::class,
166
167      /*
168      * Package Service Providers...
169      */
170
171      //
172
173      [
174          'Storage' => Illuminate\Support\Facades\Storage::class,
175          'URL' => Illuminate\Support\Facades\Url::class,
176          'Validator' => Illuminate\Support\Facades\Validator::class,
177          'View' => Illuminate\Support\Facades\View::class,
178          'Form' => Collective\Html\FormFacade::class,
179          'Html' => Collective\Html\HtmlFacade::class,
180      ],
181
182  ];
183
184 ];
```

Рис. 10. Изменения в app.php

Теперь мы можем пользоваться функционалом модуля Forms в нашем приложении. Что нам надо делать дальше? Надо создать контроллер для работы с разделами нашей книги. Для добавления новых разделов мы будем использовать форму. В Laravel для обработки формы в контроллере используются два метода. Один предназначен для вывода формы в браузер и использует метод передачи данных GET. Второй использует метод передачи данных POST, вызывается после нажатия в форме кнопки submit и отвечает за передачу данных из формы на сервер и за их обработку.

Затем в контроллере надо будет создать метод, позволяющий выводить список существующих в БД разделов и блоки для каждого раздела. Понятно, что для контроллера нам надо будет создать некое число представлений. Аналогичную работу надо будет проделать и для второго контроллера, отвечающего за работу с блоками. Для настройки всех этих действий нам надо будет разработать систему маршрутизации в нашем приложении. Поэтому перейдем к созданию контроллеров.

Создание RESTful контроллера

В прошлый раз мы с вами создали пустой контроллер и в нем добавляли свои методы. Laravel предлагает нам другой способ создания контроллера, более подходящий при работе с моделями. Сейчас мы создадим так называемый RESTful контроллер, в котором уже будут содержаться заготовки методов для выполнения стандартных действий. Кроме этих методов такой контроллер также предлагает удобную маршрутизацию.

Термин REST очень популярен сегодня не только в среде разработчиков веб – приложений, поэтому уделим несколько слов его объяснению. REST является сокращением от Representational State Transition, что можно перевести как «передача состояния представления». Это архитектурный стиль для создания взаимодействия частей распределенного приложения. Центральной частью такого приложения очень часто является веб-служба, а клиентами могут быть самые разные приложения, даже выполняющиеся на разных платформах.

Как можно обеспечить общение разнородных клиентов с одной службой? Организовать их общение по кроссплатформенному протоколу. REST организовывает взаимодействие по HTTP протоколу, используя его методы передачи данных. Метод GET используется для получения данных, метод POST – для добавления новых, метод PUT – для изменения существующих данных, и метод DELETE – для удаления. Таким образом, REST можно рассматривать как стандартизацию способов взаимодействия компонент приложения по HTTP протоколу. Во многих случаях использование REST повышает производительность приложений и упрощает их архитектуру, создавая хорошо масштабируемые приложения.

Вы уже привыкли к идее хранить в адресе GET запроса, генерируемого браузером, имена контроллера и метода контроллера и, возможно, параметры для метода. REST доводит эту идею до совершенства, позволяя упаковывать в адресную строку еще и тип SQL запроса к базе данных. Ведь очень часто действия пользователе-

лей, генерирующие запрос, требуют взаимодействия с источником данных.

После этого небольшого отступления перейдем к созданию контроллеров для нашего приложения. Создадим RESTful контроллер для работы с моделью Topic. Для этого в консольном окне надо выполнить такую команду:

```
php artisan make:controller TopicController  
      - - resource
```

Перед ключом `resource` указано два дефиса без пробела между ними. В папке `\larabook\app\Http\Controllers` будет создан файл контроллера `TopicController.php`, в котором уже будут созданы заготовки методов с комментариями, описывающими назначение каждого метода. С этими методами мы познакомимся по ходу выполнения работы.

У RESTful контроллеров, помимо наличия заготовок для требуемых методов, есть еще одно преимущество – маршрутизация для их методов выполняется намного проще. Вы должны помнить из нашего первого Laravel приложения, что для каждого метода контроллера надо прописывать свой маршрут в файле `\larabook\routes\web.php`. Для RESTful контроллера маршрутизацию для всех его методов можно указать единственной строкой в файле маршрутизации. Добавьте в файл `\larabook\routes\web.php` такую строку:

```
Route::resource('topic', 'TopicController');
```

Теперь адреса, введенные в браузер, будут в нашем приложении транслироваться в вызовы методов контроллера TopicController таким образом:

HTTP метод	Фрагмент адресной строки	Метод контроллера	Имя маршрута
GET	/topic	index	topic.index
GET	/topic/create	create	topic.create
POST	/topic	store	topic.store
GET	/topic/{photo}	show	topic.show
GET	/topic/{photo}/edit	edit	topic.edit
PUT/PATCH	/topic/{photo}	update	topic.update
DELETE	/topic/{photo}	destroy	topic.destroy

Это значит, что если вы в адресной строке укажете имя контроллера /topic и перешлете такой запрос на сервер, используя метод GET, то обработчиком этого запроса будет метод index() контроллера topic. А если вы перешлете такой запрос на сервер, используя метод POST, то в этом случае обработчиком запроса уже будет метод store() контроллера topic. И аналогично для других вариантов.

Таким образом, указав в файле маршрутизации одну строку, мы создали сразу семь маршрутов для нашего приложения. По мере его разработки мы рассмотрим использование всех методов RESTful контроллера. Если нам понадобится создавать еще какие – либо маршруты, мы сможем делать это, как и раньше – в том же файле маршрутизации \larabook\routes\web.php. При этом запомните: если вы хотите добавить в правила маршрутизации другие маршруты, то это надо сделать до строки Route::resource.

Работа с формами

Ранее мы создали папки `topic` и `block`, вложенные в `\larabook\resources\views`, для хранения представлений наших контроллеров. Начнем создавать представления для контроллера `Topic`. Сначала создадим форму, позволяющую добавлять в БД новые разделы. Эта форма будет создана в представлении `create.blade.php`, которое будет работать с методом `create()` созданного контроллера. Создаем в папке `\larabook\resources\views\topic` файл с именем `create.blade.php` и приводим его содержимое к такому виду:

```
@extends('layouts.master')

@section('title', 'Page Title')

@section('menu')
    @parent
@endsection

@section('content')
    <div class="row">
        <div class="label label - info"
style="display:inline-block;
width:100%;">{!!$page !!}
    </div>
    <div class="row">
        {!! Form::model($topic,
array('action'=>'TopicController@store')) !!}
        <div class='form - group'>
            {!! Form::label('topicnameform', 'Topic
name') !!}
            {!! Form::text('topicnameform', '',
array('class'=>'form - control')) !!}
    
```

```
</div>
<button class="btn btn - success"
        type="submit">Add Topic</button>
{!! Form::close() !!}
</div>
@endsection
```

Как мы уже говорили, для создания форм в Laravel удобно использовать модуль Forms. Мы его уже установили и поэтому используем его методы для создания формы.

Первая строка приведенного кода указывает на то, что это представление является производным от нашего базового представления 'layouts.master'.

Рассмотрим основные методы модуля Forms:

- `{!! Form::open(['action' => 'controller@method']) !!}` – открывает форму с атрибутом 'action' = 'controller@method', по умолчанию атрибут 'method' имеет значение 'post';
- `{!! Form::open(['action' => 'controller@method', 'method' => 'get']) !!}` – открывает форму с атрибутом 'action' = 'controller@method', и с атрибутом 'method' = 'get';

Вы должны помнить, что HTML тег form поддерживает для своего атрибута method только два значения: get и post. Поэтому для передачи форм с использованием методов put и delete мы будем добавлять в форму скрытый элемент с именем '_method' и в нем указывать необходимое нам значение.

Обработчик в методе open() можно указывать еще и так:

- `{!! Form::open(['url' => 'Controller/method']) !!}` – ссылкой;
- `{!! Form::open(['route' => 'route.name']) !!}` – именованным маршрутом.
- `{!! Form::open(['action' => 'controller@method', 'file' => 'true']) !!}` открывает форму с атрибутами 'action' = 'controller@method', 'method' = 'get', и 'enctype' = 'multipart/form - data'. Такая форма предназначена для выполнения upload.
- `{!! Form::model($model, ['action' => 'controller@method']) !!}` – использование метода model() создает форму типизированную моделью.

Если в такой форме имя элемента управления совпадает с именем свойства модели, то значение, занесенное в такой элемент управления, будет автоматически попадать в соответствующее свойство модели. Мы будем использовать типизированные формы.

- `{!! Form::close() !!}` закрывает форму, типизированную или не типизированную.

Конечно же, есть ряд методов для создания любых элементов управления формы.

- `Form::label()` – создает метку;
- `Form::text()` – создает текстовое поле;
- `Form::password()` – создает поле для ввода пароля;
- `Form::hidden()` – создает скрытое поле;
- `Form::file()` – создает элемент для выбора файла;

- `Form::radio()` – создает радио-кнопку;
- `Form::checkbox()` – создает флажок;
- `Form::submit()` – создает кнопку `submit`.

Кроме перечисленных здесь методов существует еще много других. Большинство из них можно вызывать с разным набором параметров. Мы будем использовать некоторые из этих методов в нашем приложении. Почитать обо всем этом подробнее вы можете по адресу: <https://laravelcollective.com/docs/5.3/html>

Теперь вы понимаете, что означает строка `{!! Form::model($topic, ['action' => 'TopicController@store']) !!}` в нашем представлении. Она создает форму,ирующую с объектом модели `Topic`, переданным в метод `model()` в переменной `$topic`. Значение атрибута `action` у созданной формы указано как метод `store()` контроллера `Topic`. Это значит, что именно метод `store()` будет обработчиком данных этой формы.

Теперь нам надо показать пользователю эту форму в ответ на активацию соответствующего пункта меню. Отображать эту форму будет метод `create()` контроллера `TopicController`. Чтобы пользователь мог вызывать этот и другие методы контроллера, надо прописать маршруты в файле маршрутизации `\larabook\routes\web.php`. Мы уже создали маршрутизацию для контроллера `TopicController`.

Давайте добавим в метод `create()` контроллера `TopicController` вызов нашего представления `create.blade.php`. Приведите код метода `create()` к такому виду:

```
public function create()
{
    $topic=new Topic;
    return view('topic.create',
        ['topic'=>$topic,'page'=>'AddTopic']);
}
```

Рассмотрим этот код. Мы создали пустой объект модели Topic в переменной \$topic. Чтобы у нас была возможность создать объект модели Topic в классе TopicController, надо перед объявлением класса в файле контроллера указать ссылку на класс модели:

```
use App\Topic;
```

В следующей строке метода create() в операторе return мы вызываем метод view(). Поговорим о параметрах этого метода. В первом параметре мы указываем в виде строки путь к тому представлению, которое хотим отобразить. Файл нашего представления с формой добавления новой страны имеет имя create.blade.php. Обратите внимание, что для вызова этого представления нам надо указать только имя файла create, без расширения blade.php. Обычно Laravel ищет представления по пути \laratest\resources\views. Но в нашем случае представление лежит по пути \laratest\resources\views\topic. Поэтому мы указываем имя вложенной папки через точку перед именем представления – 'topic.create'.

Второй параметр метода view() – это массив, в котором мы можем создать произвольное количество разных элементов, придумать им имена и передать их

в представление. В нашем случае мы передаем в представление две переменные: `topic` с объектом модели `Topic` и `page` со строковым значением. Сейчас при вводе в адресную строку браузера такого адреса:

<http://localhost/larabook/public/index.php/topic/create>

вы увидите страницу, которая отображает созданную нами форму, подобную приведенной внизу:

The screenshot shows a web browser window with the URL `localhost/larabook/public/index.php/topic/create` in the address bar. The page title is "Main Page". On the left, there is a form field labeled "Topic name" with a placeholder "Topic name". Below the form is a green button labeled "Add Topic". The top right corner of the browser window has icons for search, refresh, and star.

Рис. 11. Форма добавления разделов

Понятно, что ваше оформление может отличаться от приведенного на рисунке. Зафиксируйте на данный момент, что для вывода формы мы использовали только метод `create()` нашего RESTful контроллера. Теперь займемся обработкой данных этой формы.

Вы должны помнить, что за обработку данных, введенных в форму, отвечает метод контроллера `store()`. Мы с вами уже пользуемся правилами REST. Еще раз отметьте для себя, что эти правила универсальны и работают везде, где используется REST.

Теперь напишем код, который будет принимать введенные в форму данные, проверять их и добавлять новую запись в таблицу `Topics`. Приведем метод `store()` контроллера `TopicController` к такому виду:

```
public function store(Request $request)
{
    $topic = new Topic; //create empty model object
        //initialize the model's property topicname
        //with data entered into form's control
        //topicname
        //we have access to form controls through
        //{$request parameter
    $topic ->topicname=$request ->topicname;
    $topic ->save(); //save model into DB table
}
```

У нас очень простая модель, содержащая только два поля: идентификатор `id` и название раздела `topicname`. Значения для поля `id` генерируются автоматически, поэтому мы в форме не предусмотрели ввод значения для этого поля. Наша форма содержит только одно текстовое поле для названия раздела – `topicname`.

В строке:

```
$topic ->topicname=$request ->topicname;
```

мы заносим в свойство модели `topicname` значение, занесенное пользователем в текстовое поле `topicname` формы. Обратите внимание, что к данным, занесенным в форму, мы обращаемся через объект `$request`, автоматически переданный в наш метод.

Отобразите нашу форму в браузере, занесите в нее название какого – либо раздела и нажмите кнопку `Add Topic`. Если вы не увидели никаких сообщений об ошибках, перейдите в MySQL и выполните запрос `select * from topics`. Вы должны увидеть в таблице `topics` название добавленного раздела. Похоже, все работает? Не

совсем. Во-первых, мы не выполнили проверку введенных данных – валидацию. Во-вторых, после нажатия на кнопку Add Topic мы не понимаем, где находимся и как завершилась обработка данных формы. Давайте исправим эти недочеты.

Валидация данных формы

Вы уже помните о том, что при работе с формой обязательно надо выполнять валидацию данных, занесимых в нее пользователем. Одна из сложностей при начальном использовании Laravel заключается в том, что этот фреймворк предлагает несколько разных способов для выполнения многих технологических действий. С валидацией дело обстоит таким же образом – существуют несколько способов выполнять валидацию.

Сейчас мы с вами рассмотрим один из способов валидации данных форм. Мы выполним валидацию с помощью пакета Esensi Model Traits. Такая валидация очень проста и элегантна. Познакомиться с пакетом Esensi Model Traits подробнее можно по адресу:

<https://github.com/esensi/model>

Поскольку данный пакет не установлен по умолчанию, нам надо будет установить его с помощью Composer. Для этого в консольном окне, открытом в корневой папке приложения, надо выполнить команду:

```
composer require esensi/model 0.5.*
```

Обратите внимание, мы вновь при подключении внешнего ресурса используем composer. Выполнение этой команды может занять пару минут.

Теперь пакет Esensi Model Traits установлен. Далее нам надо заменить ссылку на класс Model в коде модели Topic. Для этого надо строку:

```
use Illuminate\Database\Eloquent\Model;
```

заменить на строку:

```
use \Esensi\Model\Model;
```

Затем надо здесь же, в классе модели, указать правила, по которым должны проверяться данные нашей формы. Добавим в класс модели еще одно свойство с именем \$rules:

```
<?php
namespace App;
use \Esensi\Model\Model;

class Topic extends Model
{
    protected $primaryKey='id';
    protected $table='topics';
    protected $fillable=['topicname',
                       'created_at', 'updated_at'];
    protected $rules=['topicname'=>
                      ['required','max:100','unique']];
}
```

Добавленное свойство содержит правила валидации для поля формы 'topicname'. Приведенные правила требуют, чтобы свойство 'topicname' модели Topic было обязательным для заполнения, не длиннее 100 символов и уникальным.

Теперь надо изменить метод store() контроллера

TopicController, добавив туда сам процесс валидации и реакцию на наличие или отсутствие ошибок валидации. Мы сделаем так, что при наличии ошибок пользователь снова увидит на экране форму добавления раздела и сообщение об ошибках. При отсутствии ошибок валидации и успешном добавлении нового раздела в БД пользователь снова увидит эту же форму, но уже с сообщением об успешном добавлении нового раздела в таблицу БД.

Приведем код метода store() контроллера TopicController к такому виду:

```
public function store(Request $request)
{
    $topic=new Topic;
    $topic - >topicname=$request - >topicname;
    //if saving to DB table failed
    if(!$topic - >save())
    {
        $err=$topic - >getErrors();
        //get errors descriptions
        return redirect() - >
        //go back to the form with error messages
        action('TopicController@create') - >
        with('errors',$err) - >withInput();
    }
    return redirect() - >
    //if saving to DB table was successful
    action('TopicController@create') - >
    with('message','New topic '.$topic -
        >topicname.' has been added with id='.
        $topic - >id.'!');
    //go back to the form with success message
}
```

Метод `$topic ->save()` мы вызываем в операторе `if()`, потому что при наличии ошибок валидации этот метод вернет `false`. В этом случае мы создадим переменную `$errors` с сообщениями о возникших ошибках и передадим ее в наше представление с формой. Поскольку ошибок валидации может быть много, эта переменная является массивом. Он автоматически сохраняется в сессии, и в представлении мы сможем достать из сессии его элементы.

Обратите внимание на вызов метода `withInput()` в случае возникновения ошибки валидации. Если у вас в форме много полей и валидацию не прошло значение в одном из полей, вы снова увидите эту форму, но не пустую. Те значения, которые прошли валидацию, будут в форме в своих элементах управления, и вам не надо будет заносить их вновь.

Сейчас наш метод `store()` переадресовывает нас снова на метод `create()`, передавая туда либо массив `$errors`, в случае возникновения ошибок валидации, либо переменную `$message` с сообщением об успешном добавлении в БД. Значит, нам надо обеспечить в представлении `create.blade.php` вывод этих переменных. Добавьте в это представление такие строки:

```

@section('content')

<div class="label label - info">{{ $page }}</div>

@if(count(session('errors')) > 0)
    <div class="alert alert - danger">
        @foreach(session('errors') -> all() as $er)
            {{$er}}<br/>
        @endforeach
    </div>
@endif

@if(session('message'))
<div class="alert alert - success" >
    {{session('message')}}
</div>
@endif

{!! Form::model($topic,
    array('action'=>'TopicController@store')) !!}
<div class='form - group'>
    {!! Form::label('topicname', 'Topic name') !!}
    {!! Form::text('topicname', '',
        array('class'=>'form - control')) !!}
</div>
<button class="btn btn - success" type="submit">
    Add Topic</button>
{!! Form::close() !!}

@endsection

```

Попробуйте сейчас выполнить добавление нового раздела – сначала намеренно допустив ошибки валидации, а затем введя корректные данные. У меня реакция формы была такой:

Урок № 6

The screenshot shows two consecutive failed attempts to add a new topic. In the first attempt, the 'Topic name' field contains extremely long text, resulting in an error message: 'The topicname may not be greater than 100 characters.' In the second attempt, the 'Topic name' field is empty, leading to the error message: 'The topicname field is required.'

The browser address bar shows: localhost/larabook/public/index.php/topic/create

Main Page Content Control

The topicname may not be greater than 100 characters.

Topic name

Very long text for topic Very long text for topic

Add Topic

The topicname field is required.

Topic name

Add Topic

Рис. 12. Данные не прошли валидацию

The screenshot shows a successful addition of a new topic. The 'Topic name' field contains the text 'MVC framework Laravel 5'. A green success message at the top of the page states: 'New topic MVC framework Laravel 5 has been added with id=5!'

The browser address bar shows: localhost/larabook/public/index.php/topic/create

Main Page Content Control

New topic MVC framework Laravel 5 has been added with id=5!

Topic name

MVC framework Laravel 5

Add Topic

Рис. 13. Данные прошли валидацию

Мы уже говорили, что это не единственный способ валидации. С другими можно познакомиться по адресу: <https://laravel.com/docs/5.3/validation>

Теперь нам надо проделать аналогичную работу для модели Block: создать RESTful контроллер, создать представление с формой для модели Block и реализовать валидацию. Начнем с создания контроллера для работы с моделью Block. Для этого в консольном окне надо выполнить такую команду:

```
php artisan make:controller BlockController
      - - resource
```

В папке \larabook\app\Http\Controllers будет создан файл контроллера BlockController.php, в котором уже будут созданы заготовки методов с комментариями, описывающими их назначения.

Создадим для этого контроллера маршрутизацию, добавив в файл \larabook\routes\web.php такую строку:

```
Route::resource('block', 'BlockController');
```

Далее создадим в папке \larabook\resources\views\block файл с именем create.blade.php и приведем его содержимое к такому виду:

```
@extends('layouts.master')

@section('menu')
@parent
@endsection

@section('content')



>{{ $page }}</div>

@if(count(session('errors')) > 0)
    <div class="alert alert - danger">
        @foreach(session('errors') ->all() as $er)
            {{$er}}<br/>
        @endforeach
    </div>
@endif

@if(session('message'))
<div class="alert alert - success" >
    {{session('message')}}
</div>
@endif


```

```

{!! Form::model($block,
    array('action'=>'BlockController@store',
        'files'=>true, 'class'=>'form')) !!}
<div class='form - group'>
    {!! Form::label('topicid', 'Select Topic',
        array('class'=>'col - md - 2')) !!}
    {!! Form::select('topicid', $topics, '',
        array('class'=>'col - md - 8')) !!}
    <a href="{{url('topic/create')}}" class=
    "btn btn - sm btn - info col - md - 2"
        type="submit">Add new Topic</a>
</div>

<div class='form - group'>
    {!! Form::label('title', 'Block Title',
        array('class'=>'col - md - 2')) !!}
    {!! Form::text('title', '', array('class'=>'col -
        md - 10')) !!}
</div>

<div class='form - group'>
    {!! Form::label('content', 'Add Content',
        array('class'=>'col - md - 2')) !!}
    {!! Form::textarea('content', '',
        array('class'=>'col - md -
        10', 'cols'=>'', 'rows'=>'')) !!}
</div>

<div class='form - group'>
    {!! Form::label('imagepath', 'Add Image',
        array('class'=>'col - md - 2')) !!}
    {!! Form::file('imagepath', '',
        array('class'=>'col - md - 10')) !!}
</div>
<button class="btn btn - success" type="submit">
    Add Block</button>
{!! Form::close() !!}
@endsection

```

Как видите, в этом представлении мы сразу предусмотрели обработку валидации таким же способом, как и в Topic контроллере. Кроме этого, форма сейчас намного интереснее. Обратите внимание на атрибут 'files=>true'. Этот атрибут метода Form::model() добавляет в создаваемый тег form атрибут enctype='multipart/form-data', что необходимо, если форма предназначена для выполнения upload. Наша форма именно такая.

Далее в форме присутствует список, в котором будут содержаться существующие разделы, чтобы пользователь мог указывать, для какого топика добавляется блок. Список создается вызовом метода Form::select(), в котором используется переменная \$topics, созданная в контроллере и инициализированная методом pluck(). Здесь же находится элемент для выбора картинки, созданный вызовом метода Form::file().

Особое внимание обратите на ссылку, расположенную сразу после списка. Ссылка, оформленная как кнопка, вызывает метод create() контроллера TopicController, позволяя пользователю добавлять новые разделы прямо с этой страницы.

Изменим определение модели Block, добавив туда правила валидации и ссылку на пакет Esensi:

```
<?php
namespace App;

use \Esensi\Model\Model;

class Block extends Model
{
    protected $primaryKey='id';
    protected $table='blocks';
    protected $fillable=array('title','topicid',
                             'content','imagesPath',
                             'created_at','updated_at');
    protected $rules=[

        'title'=>['required','max:100'],
        'topicid'=>['required'],
        'content'=>['required']
    ];
}
```

Теперь внесем изменения в созданный BlockController. Добавим требуемые ссылки на модели:

```
use App\Topic;
use App\Block;
```

и внесем изменения в метод create():

```
public function create()
{
    $block=new Block;
    $topics=Topic::pluck('topicname','id');
    return view('block.create',
    ['block'=>$block,'topics'=>$topics,'page'=>'AddBlock']);
}
```

Обратите внимание на вызов метода `pluck()` который достает из таблицы `Topics` данные для построения элемента `select` в нашей форме.

На этом этапе мы можем проверить, как выглядит наша форма, введя в браузер такой адрес:

<http://localhost/larabook/public/index.php/block/create>

У меня форма выглядит так:

Рис. 14. Форма добавления блоков

Проверим, как работает новая форма. Для этого надо изменить метод `store()` контроллера `TopicController`, приведя его к такому виду:

```
public function store(Request $request)
{
    $block=new Block;
    $fname=$request ->file('imagepath');

    if($fname != null)
    {
        $originalname=$request ->file('imagepath')
        ->getClientOriginalName();
        $request ->file('imagepath') ->
        move(public_path().'/images',$originalname);
        $block ->imagepath='/
        images/'.$originalname;
    }
}
```

```
else
{
    $block ->imagepath='';
}

$block ->title=$request ->title;
$block ->topicid=$request ->topicid;
$block ->content=$request ->content;

if(!$block ->save())
{
    $err=$block ->getErrors();
    return redirect() ->
        action('BlockController@create') ->
        with('errors',$err) ->withInput();
}
return redirect() ->
    action('BlockController@create') ->
    with('message','New block '.$block ->id.' has been added!');
}
```

Проверьте, как работает ваша форма. Занесите несколько блоков с текстовым контентом и картинками. У меня результат выглядит так:

The screenshot shows a web page with the URL `localhost/larabook/public/index.php/block/create`. The page has a header with 'Main Page' and 'Content Control'. A green banner at the top says 'New block 3 has been added!'. Below it is a form with fields: 'Select Topic' (dropdown menu showing 'Laravel 5 tutorial'), 'Block Title' (empty input field), 'Add Content' (empty input field), 'Add Image' (button with placeholder 'Выберите файл' and note 'Файл не выбран'), and a 'Add Block' button.

Рис. 15. Успешное добавление блока

Давайте подключим созданную форму добавления нового блока ко второму пункту нашего меню Content

Control. При необходимости новые разделы можно добавлять прямо из этой формы. Поэтому единственный пункт меню позволит нам добавлять и новые разделы, и новые блоки. Откройте наше мастер-представление master.blade.php из папки \larabook\resources\views\layouts и сделайте в нем такое изменение в разметке второго пункта меню:

```
<li role="presentation"
     {{ $page == 'Forms' ? 'class=active' : '' }}>
    <a href="{{ url('block/create') }}">
        Content Control</a></li>
```

Создание первой страницы

Нам предстоит создать первую страницу нашего сайта, на которой пользователь будет видеть разделы книги и блоки выбранного раздела. Как мы говорили, в левой части страницы будем отображать список разделов. При клике по какому-либо разделу в правой части страницы будем отображать все блоки этого раздела. Создаваться эта страница будет в методе index() контроллера TopicController. Приведите этот метод к такому виду:

```
public function index()
{
    $topics=Topic::all();
    $id=0;
    return view('topic.index',
        ['page'=>'home', 'topics'=>$topics,
         'id'=>$id]);
}
```

Данный метод создает и передает в свое представление две переменные: \$topics и \$id. В первой переменной, являющейся массивом, находится список разделов, полученный вызовом метода all() из таблицы topics. На основании данных из этого массива в левой колонке представления будет создаваться список из названий всех разделов. Во второй переменной сейчас находится 0, и она никак не используется в представлении. Дело в том, что это же представление будет вызываться и из другого метода этого контроллера —метода show(). Этот метод будет активироваться при клике по какому – либо из разделов в левой колонке. Тогда эта переменная будет содержать значение идентификатора того раздела, по которому кликнет пользователь.

Теперь надо создать представление с именем index.blade.php в папке \larabook\resources\views\topic. Создайте это представление и добавьте в него такой код:

```
@extends('layouts.master')
@section('menu')
@parent
@endsection

@section('content')

<div class="row">
    <div class="col - sm - 3 col - md - 3 col -
                    lg - 3 dleft">
        <ul style="list - style - type:none">
            @foreach($topics as $t)
                <li>
                    <a href=
                        "{{url('topic/'.$t ->id)}}">
                        {{$t ->topicname}}
                    </a>
                </li>
            @endforeach
        </ul>
    </div>
</div>
```

```
        @endforeach
    </ul>
</div>
<div class="col - sm - 9 col - md - 9 col -
           lg - 9 dright">
    </div>
</div>
@endsection
```

Обсудим этот код. Прежде всего, мы разделили страницу по ширине на две части в соотношении 25% к 75%. В левой части мы обрабатываем массив \$topics, переданный в представление из контроллера. В этом массиве содержится список разделов, полученный из таблицы topics, и мы создаем из его элементов список. Каждый элемент этого списка кликабельный. При клике вызывается метод show(), которому передается идентификатор кликнутого раздела. Обратите внимание, что правая, более широкая часть страницы сейчас не используется. Она будет использоваться при обращении к этому представлению из метода show(). Если вы посмотрите в таблицу адресации, приведенную на странице 31, то увидите, что вызову вида topic/id соответствует как раз метод show().

Приведите метод show() в контроллере TopicController к такому виду:

```
public function show($id)
{
    $blocks=Block::where('topicid', '=', $id)
        ->get();
    $topics=Topic::all();
    return view('topic.index', ['page'=>'Main
        page', 'topics'=>$topics, 'id'=>$id,
        'blocks'=>$blocks]);
}
```

Первая строка в этом методе выполняет запрос SELECT * FROM topics WHERE id=\$id, где \$id – идентификатор кликнутого раздела, который передается при вызове метода show(). Результат выполнения этого SQL запроса заносится в переменную \$blocks. Чтобы в контроллере был доступ к модели Block, надо перед определением класса TopicController добавить ссылку на нашу модель:

```
use App\Block;
```

Затем, так же, как и в методе index(), создается массив \$topics со списком разделов. И после этого снова вызывается то же представление index.blade.php, в которое передаются переменные \$page, \$topics, \$id и \$blocks.

Сейчас надо добавить разметку в правую часть нашего представления, где должны отображаться блоки кликнутого раздела. Это будет самая сложная и одновременно самая красивая страница нашего приложения.

Мы говорили о том, что каждый раздел содержит произвольное количество блоков. Поэтому, когда пользователь кликнет по какому – либо разделу, он должен

будет увидеть все его блоки. Мы будем выводить блоки в контейнерах с определенной структурой. Начинаться блок будет с заголовка, затем будет отображаться картинка, если она присутствует в блоке, затем будет выводиться текстовый контент блока. И, наконец, каждый блок будет содержать две кнопки: `edit` и `delete`.

Первая кнопка позволит редактировать содержимое блока. Клик по этой кнопке приведет к отображению заполненной формы, в которой будет содержаться текущий контент блока. Эта кнопка будет использовать два метода контроллера `BlockController`: `edit()` и `update()`. Первый будет отображать заполненную форму, второй – выполнять изменение блока.

Вторая кнопка позволит удалить блок. Она будет использовать метод `destroy()` контроллера `BlockController`.

По правилам REST маршрутизации обращение к методу `update()` выполняется с применением HTTP метода `put`, а обращение к методу `destroy()` – с применением метода `delete`. Поэтому каждая из этих двух кнопок будет располагаться в собственной форме, а в каждой из этих форм будет скрытый элемент для указания метода. Перейдем к рассмотрению кода, который реализует наш план.

Приведите правую часть представления к такому виду:

```

<div class="col - sm - 9 col - md - 9 col - lg - 9 pull
- right">
<! - - Check if a topic on the left was clicked - - >
@if($id != 0)
    @foreach($blocks as $b)
        <div>
            <div>
                <! - - Block's title - - >
                <h2>{{$b ->title}}</h2>
            </div>
                <! - - Check if an image exists and show
                it if it exists - - >
                @if($b ->imagePath != "")
                    <a href="{{url($b ->imagePath)}}"
                        style="float:left;
                        margin-right:20px" target="_blank"
                        class="wrap - image">
                        
                        height="150px" alt=""/>
                    </a>
                @endif
                <! - - Check if a text content exists and
                show it if it exists - - >
                <pre class="pre_text">{{$b ->content}}
                </pre>
                <! - - Form for Delete button - - >
                {!! Form::open(array('route'=>array('block.
                    destroy',$b ->id))) !!}
                    <! - - set HTTP method DELETE for
                    the form - - >
                    {{ Form::hidden('_method','DELETE') }}
                    <button class="btn btn - xs btn -
                        danger glyphicon
                        glyphicon - remove" style="float:right"
                        type="submit">
                    </button>
                {!! Form::close() !!}
                <! - - Form for Edit button - - >
                {!! Form::model($b,array('route'=>
                    array('block.update',$b ->id))) !!}

```

```

<! - - set HTTP method PUT for the
form - - >

{{ Form::hidden('_method', 'PUT') }}
<a class="btn btn - xs btn - info
glyphicon glyphicon - pencil"
style="float:right"
href="{{url('block/'.$b->id.'/edit')}}"></a>
{!! Form::close() !!}
<br><br>
</div>
@endforeach
@endif
</div>

```

Приведенные выше объяснения и комментарии в разметке вполне объясняют, как работает эта страница.

Теперь нам надо создать три метода в BlockController для обслуживания кнопок редактирования и удаления. Это методы edit(), update() и destroy(). Откройте в блокноте код этого контроллера и приведите указанные методы к такому виду:

```

public function edit($id)
{
    $block=Block::find($id);
    $topics=Topic::pluck('topicname','id');
    return view('block.edit') -
        >with('block',$block) -
        >with('topics',$topics)
        - >with('page','Main Page');
}

```

Метод find() находит в БД объект модели Block по заданному идентификатору и заносит его в переменную

\$block. Уже знакомый вам метод pluck() создает массив из всех разделов и заносит его в переменную \$topics. Затем следует вызов представления edit.blade.php. Это представление отображает пользователю форму редактирования блока, заполненную данными о выбранном блоке из переменной \$block.

Вот разметка формы редактирования блока, которую выводит в браузер метод edit(). Это представление надо создать в папке \larabook\resources\views\block под именем edit.blade.php:

```
@extends('layouts.master')

@section('menu')
@parent
@endsection

@section('content')

{!! Form::model($block, array('route'=>array('block.
update', $block - >id), 'method'=>'PUT', 'files'=>true)) !!}

<div class='form - group'>
    {!! Form::label('topicid', 'Select Topic') !!}
    {!! Form::select('topicid', $topics, $block
        - >topicid, array('class'=>'form -
control')) !!}
</div>

<div class='form - group'>
    {!! Form::label('title', 'Edit title') !!}
    {!! Form::text('title', $block - >title,
array('class'=>'form - control')) !!}
</div>
```

```

<div class='form - group'>
    {!! Form::label('content','Edit Content') !!}
    {!! Form::textarea('content', $block -
        >content, array('class'=>'form -
            control')) !!}
</div>

<div class='form - group'>
    {!! Form::label('imagepath','Edit Image') !!}
    {!! Form::file('imagepath', '',
        array('class'=>'form - control')) !!}
</div>
{!! Form::submit('Save edit block',
    array('class'=>'btn btn - primary')) !!}

```

{!!Form::close() !!}

@endsection

В этой форме вам все уже знакомо. Отметьте для себя, что в ней указан атрибут 'files'=>true, а значит, мы можем выполнять upload. Не забывайте, что эта форма будет выведена пользователю с информацией о выбранном блоке, и пользователь сможет изменить значение любого ее поля. После изменения данных и нажатия на кнопку submit активируется метод update().

```

public function update(Request $request, $id)
{
    $block=Block::find($id);
    $block - >title=$request - >title;
    $block - >content=$request - >content;
    $block - >topicid=$request - >topicid;

    //upload new image from edit form
    $fname=$request - >file('imagepath');
    if($fname != null)

```

```
{  
    $originalname=$request ->file('imagepath') ->getClientOriginalName();  
    $request ->file('imagepath') ->move(public_path().'/images', $originalname);  
    $block ->imagepath='/images/'.$originalname;  
}  
  
$block ->save();  
return redirect('topic/'.$block->topicid);  
}
```

Здесь снова метод find() находит в БД объект модели Block по заданному идентификатору и заносит его в переменную \$block. Затем свойства найденного объекта изменяются данными, выбранными из формы. Доступ к данным формы мы получаем через переменную \$request, которая передается в наш метод параметром. Если пользователь выбрал в форме редактирования картинку, то эта картинка выгружается в нашу папку images. Затем следует сохранение измененного объекта в БД и переадресация на нашу страницу.

Если же пользователь в каком – либо блоке нажмет кнопку удаления, то выполнится метод destroy(), который удалит этот блок.

```
public function destroy($id)
{
    $block=Block::find($id);
    $block->delete();
    return redirect('topic');
}
```

Здесь снова метод `find()` найдет в БД объект модели `Block` по заданному идентификатору, а метод `delete()` удалит этот объект.

Теперь можно посмотреть, как выглядит эта страница и как работают кнопки редактирования и удаления. У меня первая страница выглядит так:

The screenshot shows a web browser window with the URL `localhost/larabook/public/index.php/topic/1`. The page has a blue header bar with the text "Main Page". Below the header, there is a sidebar with links: "Laravel 5 tutorial", "Codeigniter 3 tutorial", and "PHP magic methods". The main content area has a title "Laravel Installation". Below the title is a paragraph of text about Laravel dependencies and installation steps. At the bottom of the page, there is a section titled "Creating DB tables" with a screenshot of a MySQL Workbench interface. A red circle highlights the "Edit" button in the toolbar of the screenshot.

Рис. 16. Первая страница приложения

Кнопки редактирования и удаления работают.

Расширение функционала

У вас может возникнуть такой вопрос: можно ли в RESTful контроллер добавлять другие методы, помимо тех, что вставлены туда изначально? Ответ утвердитель-

ный. Вы можете добавлять в такие контроллеры произвольные методы, необходимые для вашего приложения. Давайте сделаем такое расширение функционала контроллера в нашем приложении. Добавим на главной странице фильтр, позволяющий выбирать разделы по названиям. Когда разделов будет много, такой фильтр будет весьма полезен.

Для этого в верхней части левой колонки создадим текстовое поле, в которое будет заноситься шаблон для поиска, и кнопку рядом с этим полем. Все это будет в форме. Для обслуживания этой формы добавим в контроллер TopicController метод search(). Если вам не нравится такое имя метода, назовите его, как хотите.

Сначала добавим в представление index.blade.php из папки \larabook\resources\views\topic, в левую часть, где выводится список разделов, такую разметку:

```
<div class="col - sm - 3 col - md - 3 col - lg -  
3 pull - left">  
    {!! Form::open(array('action'=>'TopicController@  
        search', 'class'=>'form')) !!}  
    <div class="input - group">  
        {!! Form::text('searchform', '',  
            array('class'=>'form - control',  
            'placeholder'=>'Enter topic')) !!}  
        <span class="input - group - btn">  
            <button class="btn btn - success btn -  
                secondary"  
                type="submit">Search</button>  
        </span>  
    </div>  
    {!! Form::close() !!}
```

```

<ul style="list - style - type:none">
@foreach($topics as $t)
    <li>
        <a href="{{url('topic/'.$t -
            >id)}}">{{{$t ->topicname}}}</a>
    </li>
@endforeach
</ul>
</div>

```

В качестве обработчика этой формы указан метод search() контроллера TopicController, которого у нас пока нет. Добавим в контроллер этот метод:

```

use App\Bloapublic function search(Request $request)
{
    $search=$request ->searchform;
    $search='%'.$search.'%';
    $topics=Topic::where('topicname','like',
        $search) ->get();
    return view('topic.index',['page'=>'Main
        Page','topics'=>$topics,'id'=>0]);
}

```

Здесь все просто. Создается шаблон для функции like и выполняется SQL запрос SELECT * FROM topics WHERE topicname like '%'.\$search.'.%. По результатам этого запроса строится список разделов в левой части страницы. Конечно же, для этого метода надо добавить маршрут в файле larabook/routes/web.php:

```

Route::post('topic/search', 'TopicController@search');
Route::resource('topic', 'TopicController');
Route::resource('block', 'BlockController');

```

Теперь первая страница нашего сайта у меня выглядит так:

Main Page Content Control

Enter topic Search

Laravel 5 tutorial
Codeigniter 3 tutorial
PHP magic methods

Laravel Installation

For managing dependencies, Laravel uses composer. Make sure you have a Composer installed on your system before you install Laravel. Step 1 – Visit the following URL and download composer to install it on your system.
<https://getcomposer.org/download/> Step 2 – After the Composer is installed, check the installation by typing the Composer command in the command prompt as shown in the following screenshot. Composer Step 3 – Create a new directory anywhere in your system for your new Laravel project. After that, move to path where you have created the new directory and type the following command there to install Laravel. `composer create-project laravel/laravel --prefer-dist` Step 4 – The above command will install Laravel in the current directory. Start the Laravel service by executing the following command.

Creating DB tables

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema. If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve. The Laravel Schema facade provides database agnostic support for creating and manipulating tables across all of

Рис. 17. Фильтр для выбора разделов

Аутентификация

Вы, конечно же, обратили внимание, что мы до сих пор избегали вопросов об аутентификации и авторизации. С другой стороны, вы видели в базе данных таблицы `Users` и `Password_resets`, названия которых говорят об их назначении. Пришло время поговорить о том, как регистрировать и авторизировать пользователей в Laravel приложении.

Laravel по умолчанию включает в свой состав встроенную систему аутентификации. Функционал этой системы подходит для подавляющего большинства приложений в том виде, как он есть. Но при необходимости его можно дорабатывать. В папке `app` находится модель `User`, которая описывает пользователей. Кроме этой модели, в пространстве имен `App\Http\Controllers\Auth` также уже встроено несколько контроллеров:

- `RegisterController` – отвечает за регистрацию новых пользователей;
- `LoginController` – отвечает за аутентификацию;
- `ForgotPasswordController` – отвечает за email пересылки при восстановлении пароля;
- `ResetPasswordController` – содержит логику восстановления пароля.

Если в вашем приложении нужны аутентификация и авторизация, вы можете быстро создать все необходимые представления и маршруты для них, выполнив такую artisan команду:

```
php artisan make:auth
```

Эта команда создаст все представления, маршруты и еще один контроллер HomeController, в котором выполняется настройка переадресации после регистрации и после входа. Кроме того, эта же команда создаст папку с именем larabook/resources/views/layouts, а в ней – представление с именем app.blade.php, которое является базовым для приложения и уже включает в себя формы регистрации и входа.

Для вас это звучит странно, потому что у нас уже создана эта папка, а в ней – базовое представление master.blade.php. Дело в том, что команду «`php artisan make:auth`» надо выполнять в самом начале создания нового приложения, а затем уже строить свое приложение вокруг созданного базового представления. Нам же надо будет просто изменить для нашего приложения базовое представление, чтобы оно работало с созданной системой аутентификации. Выполним эту работу.

Нам будет проще добавить в только что созданное представление `app.blade.php` два своих пункта меню и настройки на Bootstrap, чем добавлять в наше представление `master.blade.php` весь код, необходимый для внедрения аутентификации и авторизации. Поэтому откройте в блокноте представление `app.blade.php` и добавьте в него отмеченные строки из `master.blade.php`:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf - 8">
    <meta http - equiv="X - UA - Compatible"
          content="IE=edge">
    <meta name="viewport" content="width=device - width,
                                   initial - scale=1">

    <!-- CSRF Token -->
    <meta name="csrf - token" content=
          "{{ csrf_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Styles -->
    <link href="{{asset('css/app.css')}}" rel="stylesheet">
    <link href="{{asset('css/style.css')}}" rel="stylesheet">
    <link href="{{asset('css/bootstrap.min.css')}}" rel="stylesheet">
    <link href="{{asset('css/bootstrap - theme.min.
                           css')}}" rel="stylesheet">
    <!-- Latest compiled and minified
        JavaScript -->
    <script type="text/javascript" src="{{asset('js/
                                              bootstrap.min.js')}}"></script>
    <script type="text/javascript" src="{{asset('js/
                                              jquery - 3.0.0.js')}}"></script>
    <script type="text/javascript" src="{{asset('js/app.
                                              js')}}"></script>

    <!-- Scripts -->
    <script>
        window.Laravel = <?php echo json_encode([
            'csrfToken' => csrf_token(), ]); ?>
    </script>
</head>

```

```
<body>
  <div id="app">
    <nav class="navbar navbar - default navbar -
           static - top">
      <div class="container">
        <div class="navbar - header">

          <! - - Collapsed Hamburger - - >
          <button type="button" class="navbar -
           toggle collapsed"
                  data - toggle="collapse" data -
                  target="#app - navbar - collapse">
            <span class="sr - only">Toggle
              Navigation</span>
            <span class="icon - bar"></span>
            <span class="icon - bar"></span>
            <span class="icon - bar"></span>
          </button>

          <! - - Branding Image - - >
          <a class="navbar - brand" href="{{
              url('/') }}>
            {{ config('app.name', 'Laravel') }}>
          </a>
        </div>

        <div class="collapse navbar - collapse"
             id="app - navbar - collapse">
          <! - - Left Side Of Navbar - - >
          <ul class="nav navbar - nav">
            <li role="presentation" >
              <a href="{{url('topic')}}">
                Main Page</a>
            </li>
            <li role="presentation" >
              <a href="{{url('block/
                create')}}">Content
                Control</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </div>
</body>
```

```

<! - - Right Side Of Navbar - - >
<ul class="nav navbar - nav navbar - right">
    <! - - Authentication Links - - >
    @if (Auth::guest())
        <li><a href="{{ url('/login') }}>Login</a></li>
        <li><a href="{{ url('/register') }}>Register</a>
    </li>
    @else
        <li class="dropdown">
            <a href="#" class="dropdown - toggle" data - toggle="dropdown" role="button" aria - expanded="false">
                {{ Auth::user() ->name }} <span class="caret"></span>
            </a>

            <ul class="dropdown - menu" role="menu">
                <li>
                    <a href="{{ url('/logout') }}" onclick="event.preventDefault();
                        document.getElementById('logout - form').submit();">
                        Logout
                    </a>

                    <form id="logout - form" action="{{ url('/logout') }}" method="POST" style="display: none;">
                        {{ csrf_field() }}
                    </form>
                
```

```

        </li>
    </ul>
</li>
@endif
</ul>
</div>
</div>
</nav>

@yield('content')
</div>

<! - - Scripts - - >
<! - - <script src="/js/app.js"></script> - - >
</body>
</html>

```

Как видите, изменений совсем немного. Теперь нам надо изменить имя базового представления в четырех представлениях для контроллеров. Откройте представления create.blade.php и edit.blade.php в папке larabook\resources\views\block и в первой строке обоих представлений:

```
@extends('layouts.master')
```

замените на

```
@extends('layouts.app')
```

Аналогичные изменения сделайте для представлений create.blade.php и index.blade.php в папке larabook\resources\views\topic. Все – система аутентификации и авторизации в нашем приложении настроена и готова к работе. Посмотрим, как выполняется регистрация

пользователей и их вход в систему. Запустите наше приложение. Сейчас на главной странице вы видите два новых пункта меню: Login и Register:

Laravel Main Page Content Control

Login Register

Register

Name

E-Mail Address

Password

Confirm Password

Register

Рис. 18. Новый вид главной страницы

Зарегистрируйте нового пользователя, используя уже готовую форму:

localhost/larabook/public/index.php/topic/1

Laravel Main Page Content Control

Login Register

Enter topic Search

Laravel Installation

For managing dependencies, Laravel uses composer. Make sure you have a Composer installed on your system before you install Laravel. Step 1 – Visit the following URL and download composer to install it on your system: <https://getcomposer.org/download/> Step 2 – After the Composer is installed, check the installation by typing the Composer command in the command prompt as shown in the following screenshot. Composer Step 3 – Create a new directory anywhere in your system for your new Laravel project. After that, move to path where you have created the new directory and type the following command there to install Laravel: `composer create-project laravel/laravel --prefer-dist` Step 4 – The above command will install Laravel in the current directory. Start the Laravel service by executing the following command: `php artisan serve`

Creating DB tables

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema. If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve. The Laravel Schema

Рис. 19. Форма регистрации

Валидация данных этой формы и добавление зарегистрированных пользователей в БД в таблицу Users тоже уже настроены.

После входа в систему зарегистрированного пользователя в браузер выводится такое сообщение, а меню

теперь отображает имя вошедшего пользователя.



Рис. 20. Успешный вход на сайт

В качестве примера использования аутентификации и авторизации давайте сделаем так, чтобы при попытке активации второго пункта меню нашего сайта незарегистрированным пользователем, такой пользователь пересыпался бы на страницу входа. Для этого надо добавить несколько строк в метод `create()` контроллера `BlockController`:

```
public function create()
{
    if (!Auth::check())
    {
        return redirect('login');
    }
    $block=new Block;
    $topics=Topic::pluck('topicname','id');
    return view('block.create',
    ['block'=>$block,'topics'=>$topics,'page'=>'Forms']);
}
```

Чтобы проверить, вызван ли метод `create()` зарегистрированным пользователем, мы используем статический метод `check()` класса `Auth`. Если этот метод вернет `false`, значит, вызов выполнен от имени не зарегистрированного пользователя. В этом случае пользователь будет перенаправлен на страницу входа. Для того чтобы у нас

был доступ к классу Auth, нам надо перед определением класса контроллера добавить ссылку на пространство имен, в котором определен класс Auth:

```
use Illuminate\Support\Facades\Auth;
```

Попробуйте сейчас войти во второй пункт меню, если вы не выполнили вход на сайт. Вы будете переадресованы на страницу входа.

Подведение итогов

Вы познакомились с самым популярным языком для веб-разработки. В ходе наших уроков вы научились писать код как на чистом PHP, так и с помощью фреймворков. Сейчас, когда уроки пройдены, вы должны запомнить один важный совет: ваша учеба на этом не закончилась. PHP постоянно развивается, регулярно создаются новые PHP фреймворки и выпускаются новые версии существующих. Для того, чтобы уметь работать со всеми этими инструментами, надо продолжать учебу. Теперь вы будете делать это уже самостоятельно.



Урок №6

Создание web-приложений, исполняемых на стороне сервера при помощи языка программирования PHP и технологии AJAX

© Александр Геворкян
© Компьютерная Академия «Шаг»
www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопрограммирования, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.