MVC фреймворк Laravel

Laravel

Laravel — бесплатный веб-фреймворк с открытым кодом, предназначенный для разработки с использованием архитектурной модели MVC.

https://laravel.com/

https://laravel.ru/

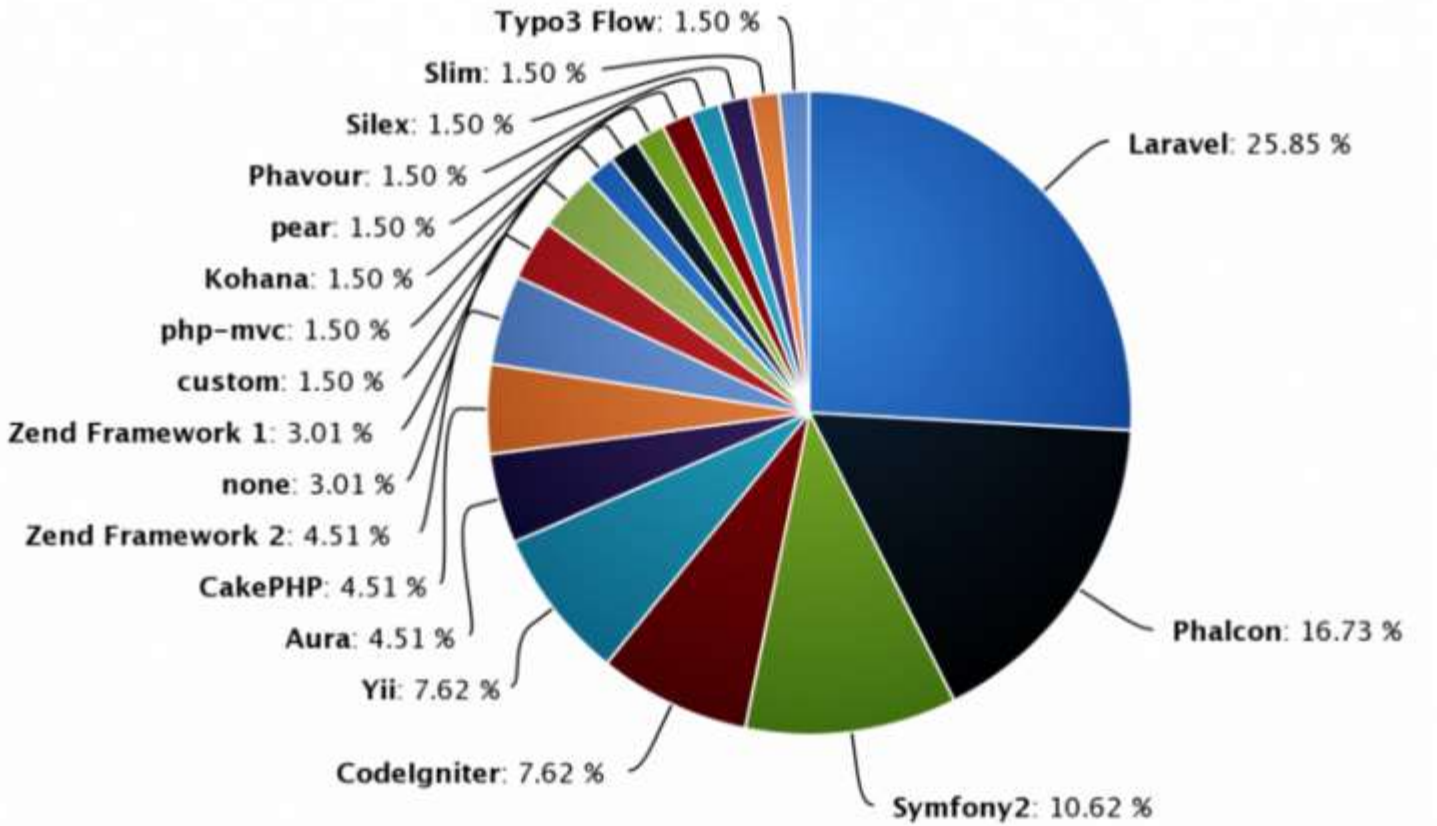
Laravel

* Laravel был создан Taylor Otwell как более функциональная альтернатива CodeIgniter, который не предусматривал различные дополнительные функции.
* Первый бета-релиз Laravel стал доступен 9 июня 2011 года, а Laravel 1 вышел в этом же месяце. Laravel 1 включал в себя встроенную поддержку аутентификации, локализации, модели, представления, сессии, маршрутизации и другие механизмы.
* Laravel 2 был выпущен в сентябре 2011 года. Основные новые функции включают в себя поддержку контроллеров, которые сделали фреймворк полностью MVC-совместимым, встроенную поддержку для инверсии управления(*Inversion of Control*, *IoC*), и систему шаблонов Blade.
* Версия Laravel 3 была выпущена в феврале 2012 года с набором новых функций, включая интерфейс командной строки (CLI) под именем "Artisan", встроенную поддержку нескольких систем управления базами данных, поддержку контроля версий баз данных в виде системы миграций, обработку событий. Выпуск Laravel 3 получил значительное увеличение числа пользователей, что повлияло на дальнейший рост его популярности.

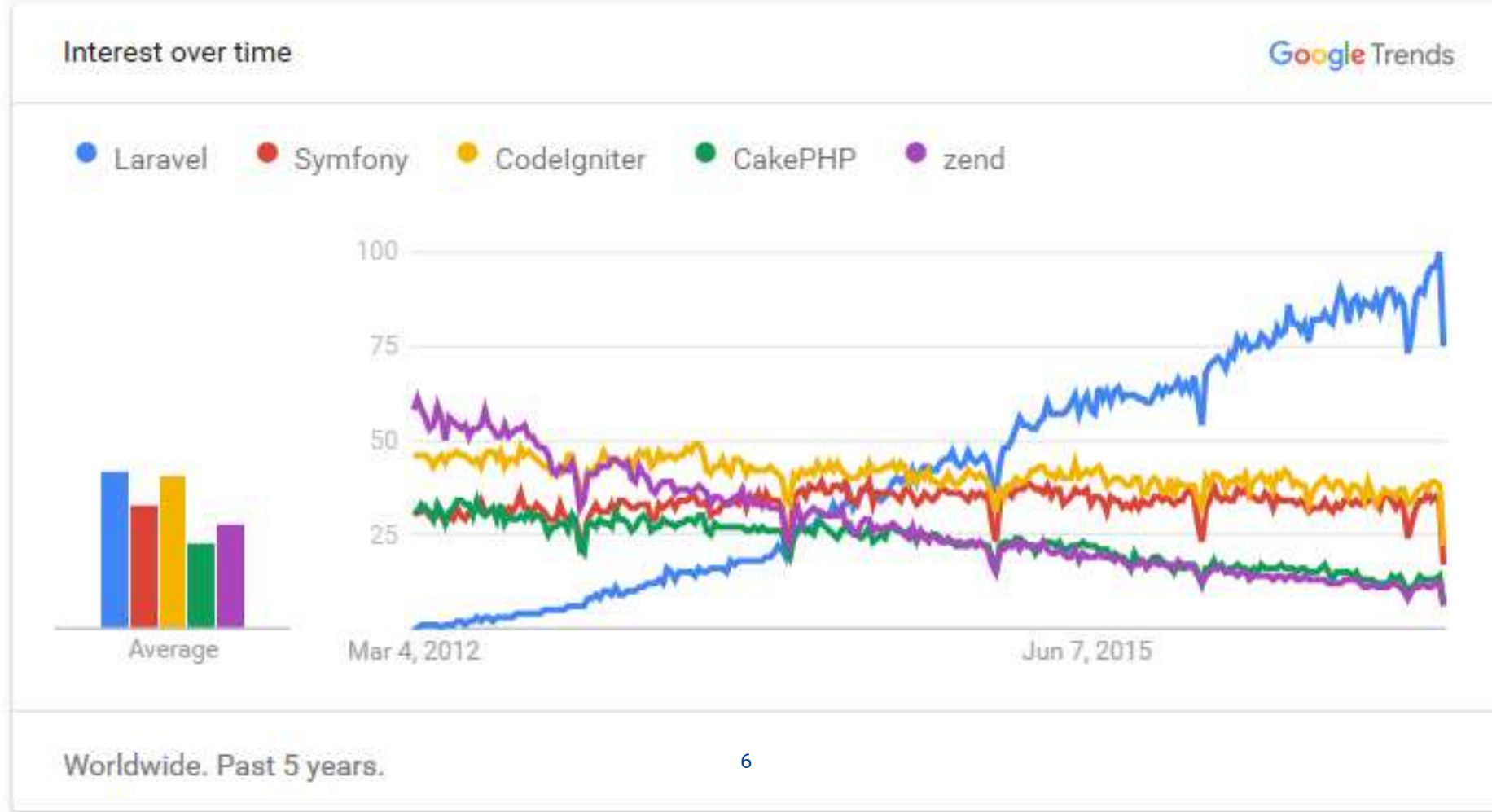
Laravel

* Laravel 4, под кодовым названием Illuminate, был выпущен в мае 2013 года. Была выполнена полная переработка фреймворка, преобразовавшая его в набор отдельных пакетов, распространяемых через Composer, который служит в качестве менеджера пакетов на уровне приложений. Другие нововведения в релизе Laravel 4 включают возможности инициализации базы данных(seeding), поддержку отложенного удаления записей из базы данных(мягкое удаление), поддержку очередей сообщений, встроенную поддержку отправки различных типов электронных сообщений.
* Laravel 5 был выпущен в феврале 2015 года. Laravel 5 включает в себя: поддержку для планирования периодически выполняемых задач (пакет Scheduler); абстракцию для работы с файловой системой (пакет Flysystem - содержит простые в использовании драйвера для работы с локальными файловыми системами, Amazon S3 и Rackspace Cloud Storage); добавлена поддержка нескольких препроцессоров CSS(less, sass) и JavaScript(coffee script).
* Текущая версия Laravel – 5.8
* В апреле 2015 был выпущен Lumen - микрофреймворк на основе компонентов Laravel.

Laravel


Typo3 Flow: 1.50 %
Slim: 1.50 %
Silex: 1.50 %
Phavour: 1.50 %
pear: 1.50 %
Kohana: 1.50 %
php-mvc: 1.50 %
custom: 1.50 %
Zend Framework 1: 3.01 %
none: 3.01 %
Zend Framework 2: 4.51 %
CakePHP: 4.51 %
Aura: 4.51 %
Yii: 7.62 %
CodeIgniter: 7.62 %
Symfony2: 10.62 %
Phalcon: 16.73 %
Laravel: 25.85 %

Laravel


Interest over time
Google Trends
Laravel
Symfony
CodeIgniter
CakePHP
zend
100
75
50
25
Average
Mar 4, 2012
Jun 7, 2015
Worldwide. Past 5 years.

Laravel

Установка

Laravel использует **Composer** для управления зависимостями. Поэтому сначала необходимо установить Composer, а затем Laravel.
Composer - это пакетный менеджер уровня приложений для PHP, который предоставляет средства по управлению зависимостями в PHP-приложении(аналоги: npm для Node.js, bundler для ruby).
Composer работает через интерфейс командной строки и позволяет пользователям устанавливать PHP-приложения, которые доступны на **packagist.org**, который является его основным репозиторием, где содержатся все доступные пакеты.
Для установки Laravel необходимо иметь:

PHP >= 5.4
MCrypt (расширение для PHP)
OpenSSL (расширение для PHP)
Mbstring (расширение для PHP)
Tokenizer (расширение для PHP)

Установка может быть выполнена с использованием команды :
composer global require "laravel/installer=~1.1"

Laravel

Установка

Права доступа

Для Laravel может потребоваться, чтобы у веб-сервера были права на запись в папки ***storage*** и ***vendor***.

Laravel поставляется вместе с файлом public/.htaccess, который настроен для обработки без указания в URL index.php. Необходимо включить модуль **mod_rewrite** в том случае, если в качестве веб-сервера используется Apache:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Laravel

Установка Homestead

Laravel Homestead — официальный предустановленный контейнер **Vagrant**, предоставляющий среду разработки без необходимости установки PHP, HHVM, веб-сервера и любого другого программного серверного обеспечения.

Vagrant — свободное и открытое программное обеспечение для создания и конфигурирования виртуальной среды разработки, и является обёрткой для программного обеспечения виртуализации, например, **VirtualBox**, и средств управления конфигурациями, такими как **Chef**.

Homestead запускается на ОС Windows, Mac и Linux, и включает в себя веб-сервер Nginx, PHP 5.6, MySQL, Postgres, Redis, Memcached и все необходимое для разработки Laravel-приложений.

Прежде чем запустить Homestead, необходимо установить VirtualBox и Vagrant. Эти программные пакеты предоставляют простые в использовании визуальные инсталляторы для всех популярных операционных систем.

После этого достаточно выполнить команды:

vagrant box add laravel/homestead

vagrant up

Laravel

Структура Laravel приложения

Корневой каталог свежеустановленного Laravel содержит ряд папок:

* Папка **app**, содержит код ядра приложения.
* Папка **bootstrap** содержит несколько файлов, которые загружают фреймворк и настраивают автозагрузку.
* Папка **config**, содержит все конфигурационные файлы приложения.
* Папка **database** содержит миграции и классы для наполнения начальными данными для БД.
* Папка **public** содержит фронт-контроллер и контент (изображения, JavaScript, CSS, и т.д.).
* Папка **resources** содержит шаблоны, сырой контент (LESS, SASS, CoffeeScript) и «языковые» файлы.
* Папка **storage** содержит скомпилированные Blade-шаблоны, файл-сессии, кэши файлов и другие файлы, создаваемые фреймворком.
* Папка **tests** содержит автотесты.
* Папка **vendor** содержит Composer-зависимости(используемые сторонние пакеты).

Laravel

Каталог app

В каталоге app находится ряд дополнительных каталогов, таких как Console, Http, Providers и тд.

Каталоги Console и Http предоставляют API «ядра» приложения.

Протокол HTTP и командная строка — это механизмы взаимодействия с приложением, но они не содержат логики приложения.

Каталог Console содержит все Artisan-команды, а каталог Http содержит контроллеры, фильтры и запросы.

В папке Commands, хранятся команды для приложения. Команды представляют собой задания, которые могут быть обработаны приложением в порядке очереди, а также задачи, которые можно запустить синхронно в рамках прохождения текущего запроса.

В папке Events, хранятся классы событий.

Laravel

Каталог app

Папка Handlers содержит классы обработчиков команд и событий. Обработчики получают команду или событие и выполняют логику в ответ на эту команду или возникновение события.

Папка Services содержит ряд «вспомогательных» служб, необходимых для работы приложения. Например, включённая в Laravel служба Registrar отвечает за проверку и создание новых пользователей приложения. Другой пример — службы для взаимодействия с внешними API.

Папка Exceptions содержит обработчики исключений приложения.

Laravel

* Маршрутизация

Большинство маршрутов (routes) приложения будут определены в файле **app/Http/routes.php**, который загружается классом App\Providers\RouteServiceProvider. В Laravel, простейший маршрут принимает URI (путь) и функцию-замыкание:

```
Route::get('/', function()
{
  return 'Hello World';
});
Route::post('foo/bar', function()
{
  return 'Hello World';
});
 Route::match(['get', 'post'], '/', function()
{
  return 'Hello World';
});
Route::any('foo', function()
{
  return 'Hello World';
});
```

Laravel

* Маршрутизация. Параметры маршрутов

```
Route::get('user/{id}', function($id)
{
  return 'User '.$id;
});
```

Необязательные параметры маршрута

```
Route::get('user/{name?}', function($name = null)
{
  return $name;
});
```

Необязательные параметры со значением по умолчанию

```
Route::get('user/{name?}', function($name = 'John')
{
  return $name;
});
```

Маршруты с соответствием пути регулярному выражению

```
Route::get('user/{name}', function($name)
{
  //
})
->where('name', '[A-Za-z]+');
```

Laravel

Маршрутизация. Именованные маршруты

Именованные маршруты позволяют удобно генерировать URL-адреса и делать переадресацию на конкретный маршрут. Вы можете задать имя маршруту с помощью ключа массива as:

```
Route::get('user/profile', ['as' => 'profile', function()
{
  //
}]);
```

Также можно указать имена маршрутов для *действий* контроллера:

```
Route::get('user/profile', [
  'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

После этого можно использовать имя маршрута при генерации URL или создании переадресации:

```
$url = route('profile');
$redirect = redirect()->route('profile');
```

Метод currentRouteName() возвращает имя маршрута, обрабатывающего текущий запрос:

```
$name = Route::currentRouteName();
```

Laravel

* Маршрутизация. Пространства имён.

Возможно использовать параметр namespace в массиве групповых атрибутов для указания пространства имён для всех контроллеров в группе:

```
Route::group(['namespace' => 'Admin'], function()
{
  // Контроллеры в пространстве имён "App\Http\Controllers\Admin"

  Route::group(['namespace' => 'User'], function()
  {
    // Контроллеры в пространстве имён "App\Http\Controllers\Admin\User"
  });
});
```

Laravel

* Маршрутизация. Пример routes.php.

```
Route::prefix('posts')->namespace('Front')->group(function () {
   Route::name('posts.display')->get('{slug}', 'PostController@show');
   Route::name('posts.tag')->get('tag/{tag}', 'PostController@tag');
   Route::name('posts.search')->get('', 'PostController@search');
   Route::name('posts.comments.store')->post('{post}/comments',
                                                'CommentController@store');
   Route::name('posts.comments.comments.store')->post(
     '{post}/comments/{comment}/comments', 'CommentController@store'
   );
   Route::name('posts.comments')->get('{post}/comments/{page}',
                                    'CommentController@comments');
});

Route::resource('comments', 'Front\CommentController', [
   'only' => ['update', 'destroy'],
   'names' => ['destroy' => 'front.comments.destroy']
]);
```

Laravel

* Middleware

HTTP-посредники (*middleware*) предоставляют удобный механизм для фильтрации HTTP-запросов приложения.

Например, в Laravel есть посредник для проверки авторизации пользователя. Если пользователь не авторизован, посредник перенаправит его на экран входа в систему. В противном случае, посредник позволит запросу пройти в приложение.

Посредники нужны не только для авторизации. CORS-посредник(Cross-origin resource sharing) может пригодиться для добавления особых заголовков ко всем ответам в приложении. А посредник логов может зарегистрировать все входящие запросы.

В Laravel есть стандартные посредники, включая посредники для обслуживания, авторизации, CSRF-защиты(Cross Site Request Forgery) и многие другие.

Все *middleware* расположены в каталоге app/Http/Middleware.

Laravel

* Middleware. Создание посредника

Одним из примеров посредника в Laravel - посредник для проверки авторизации пользователя. Если пользователь не авторизован, посредник перенаправит его на страницу входа в систему. В противном случае, посредник позволит запросу пройти в приложение:

```
public function handle($request, Closure $next)
    {
        $user = $request->user();
        if ($user && $user->role === 'admin') {
            return $next($request);
        }
        return redirect()->route('home');
    }
```

Laravel

* Middleware. Создание посредника

```
<?php namespace App\Http\Middleware;
use Closure;
class OldMiddleware {
  /**
  * Выполнение фильтра запроса.
  *
  * @param  \Illuminate\Http\Request  $request
  * @param  \Closure  $next
  * @return mixed
  */
  public function handle($request, Closure $next)
  {
    if ($request->input('age') < 200)
    {
      return redirect('home');
    }

    return $next($request);
  }
}
```

Laravel

* Middleware. Создание посредника

То, в какой момент сработает посредник — до или после запроса, зависит от его реализации. Например, этот посредник выполнит некоторую задачу **перед** тем как запрос будет обработан приложением:

```
<?php namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware implements Middleware {

  public function handle($request, Closure $next)
  {
    // Perform action

    return $next($request);
  }
}
```

Laravel

* Middleware. Создание посредника

Этот посредник выполнит задачу **после** того, как запрос будет обработан приложением:

```
<?php namespace App\Http\Middleware;

use Closure;

class AfterMiddleware implements Middleware {

  public function handle($request, Closure $next)
  {
    $response = $next($request);

    // Perform action

    return $response;
  }
}
```

Laravel

Middleware. Регистрация посредника

- Глобальный посредник

Если необходимо, чтобы посредник запускался для каждого HTTP-запроса в приложении, надо добавить его в свойство $middleware класса app/Http/Kernel.php.

- Назначение посредника для маршрутов

Если необходимо назначить посредника для конкретных маршрутов, то сначала надо добавить сокращённый ключ посредника в класс app/Http/Kernel.php. По умолчанию свойство $routeMiddleware этого класса содержит записи посредников Laravel. Чтобы добавить собственный посредник, необходимо просто добавить его к этому списку и присвоить ему ключ.

Когда посредник определён в HTTP-ядре, возможно использовать ключ middleware в массиве параметров маршрута:

```
Route::get('admin/profile', ['middleware' => 'auth', function()
{
    //
}]);
```

Laravel

* Контроллеры

Контроллеры могут группировать связанную с обработкой HTTP-запросов логику в отдельный класс. Контроллеры хранятся в папке app/Http/Controllers.

Пример простейшего класса контроллера(контроллер наследует базовый класс контроллера, встроенный в Laravel)

```
<?php namespace App\Http\Controllers;
    use App\Http\Controllers\Controller;
    class UserController extends Controller {
      /**
      * Показать профиль данного пользователя.
      *
      * @param  int  $id
      * @return Response
      */
      public function show ($id) {
        return view('user.profile', ['user' => User::findOrFail($id)]);
      }
    }
```

Маршрут для этого контроллера должен выглядеть следующим образом:

Route::get('user/{id}', 'UserController@show');

Laravel

* Контроллеры. RESTful-контроллеры ресурсов

Контроллеры ресурсов упрощают построение RESTful-контроллеров, работающих с ресурсами. Например, возможно создать контроллер, обрабатывающий фотографии, хранимые приложением. Можно быстро создать контроллер с помощью Artisan-команды make:controller:

php *artisan make:controller PhotoController*

Теперь мы можем зарегистрировать маршрут контроллера ресурса:

Route::resource('photo', 'PhotoController');

Этот единственный вызов создаёт множество маршрутов для обработки различных RESTful-действий на ресурсе photo. Сам сгенерированный контроллер уже имеет методы-заглушки для каждого из этих *действий* с комментариями о том, какие типы запросов они обрабатывают.

Laravel

* Контроллеры. RESTful-контроллеры ресурсов

***Действия*, обрабатываемые контроллером ресурсов**

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

Laravel

* Контроллеры. RESTful-контроллеры ресурсов

Настройка маршрутов ресурсов

Возможно указать подмножество *действий*, которые должны обрабатываться по маршруту:

```
Route::resource('photo', 'PhotoController',    ['only' => ['index', 'show']]);

Route::resource('photo', 'PhotoController',
    ['except' => ['create', 'store', 'update', 'destroy']]);
```

По умолчанию все *действия* контроллера ресурсов имеют имена маршрутов, но возможно переопределить эти имена, передав массив names вместе с остальными параметрами:

```
Route::resource('photo', 'PhotoController',
    ['names' => ['create' => 'photo.build']]);
```

Laravel

* Контроллеры. RESTful-контроллеры ресурсов

Обработка контроллеров вложенных ресурсов

Необходимо использовать «точечную» нотацию в объявлении маршрута для контроллеров «вложенных» ресурсов:

Route::resource('photos.comments', 'PhotoCommentController');

Этот маршрут зарегистрирует «вложенный» ресурс, к которому можно обратиться по такому URL: **photos/{photos}/comments/{comments}**.

```
class PhotoCommentController extends Controller {
    /**
     * Показать определённый комментарий фото.
     *
     * @param  int  $photoId
     * @param  int  $commentId
     * @return Response
     */
    public function show($photoId, $commentId)
    {
      //
    }
}
```

Laravel

* Запросы и ввод

Получение экземпляра текущего HTTP-запроса возможно через ***внедрение зависимости***. Для этого необходимо использовать экземпляр класса Request в конструкторе или методе контроллера. Экземпляр текущего запроса будет автоматически внедрён сервис-контейнером:

```
<?php namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

 /**
  * Сохранить нового пользователя.
  *
  * @param  Request  $request
  * @return Response
  */
 public function store(Request $request)
 {
   $name = $request->input('name');

   //
 }
}
```

Laravel

* Запросы и ввод

Если метод контроллера также ожидает ввода из параметров маршрута, необходимо перечислить аргументы маршрута после других зависимостей:

```
<?php namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

  /**
  * Обновить указанного пользователя.
  *
  * @param  Request  $request
  * @param  int  $id
  * @return Response
  */
  public function update(Request $request, $id)
  {
    //
  }

}
```

Laravel

* Запросы и ввод

Возможно получить доступ ко всем введённым данным из экземпляра Illuminate\Http\Request, используя несколько простых методов. При этом не важно какой тип HTTP-запроса был использован (GET, POST и т.д.) — методы работают одинаково для любого из них.

Получение переменной

```
$name = Request::input('name');
```

Получение переменной или значения по умолчанию, если переменная не была передана

```
$name = Request::input('name', 'Sally');
```

Была ли передана переменная?

```
if (Request::has('name'))
{
  //
}
```

Laravel

* Запросы и ввод

Получение всех переменных запроса

```
$input = Request::all();
```

Получение некоторых переменных

```
$input = Request::only('username', 'password');

$input = Request::except('credit_card');
```

При работе с переменными-массивами можно использовать точечную запись для доступа к их элементам:

```
$input = Request::input('products.0.name')
```

Laravel

* Запросы и ввод

Получение объекта загруженного файла

```
$file = Request::file('photo');
```

Определение успешной загрузки

```
if (Request::hasFile('photo'))
{
  //
}
```

Метод file() возвращает объект класса Symfony\Component\HttpFoundation\File\UploadedFile, который в свою очередь наследует стандартный PHP-класс SplFileInfo и предоставляет множество методов для работы с файлами.

Прошёл ли загруженный файл проверку?

```
if (Request::file('photo')->isValid())
{
  //
}
```

Перемещение загруженного файла

```
Request::file('photo')->move($destinationPath);

Request::file('photo')->move($destinationPath, $fileName);
```

Laravel

* Запросы и ввод

Информация о запросе

Класс Request содержит множество методов для изучения входящего в приложение запроса. Ниже представлено несколько примеров:

Получение URI (пути) запроса

```
$uri =$request->path();
```

Сделан ли запрос через AJAX?

```
if ($request->ajax()){
  //
}
```

Получение метода запроса

```
$method = $request->method();
  if ($request->isMethod('post')){
  //
}
```

Соответствует ли запрос маске пути?

```
if ($request->is('admin/*')){
  //
}
```

Получение URL запроса: `$url = Request::url();`

Laravel

* Представления

Представления (*views*) содержат HTML-код, передаваемый приложением. Это удобный способ разделения бизнес-логики и логики отображения информации. Представления находятся в каталоге resources/views.

Простое представление выглядит примерно так:

```
<!-- Представление resources/views/greeting.php -->
<html>
 <body>
  <h1>Привет, <?php echo $name; ?></h1>
 </body>
</html>
```

Laravel

* Представления

Такое представление можно вернуть в браузер примерно так:

```
Route::get('/', function()
{
  return view('greeting', ['name' => 'James']);
});
```

Первый параметр, переданный вспомогательной функции view, соответствует имени файла представления в каталоге resources/views. Вторым параметром, переданным функции, является массив данных, которые будут доступны для представления.

Если представление сохранено в resources/views/admin/profile.php, оно должно быть возвращено так:

```
return view('admin.profile', $data);
```

Laravel

* Представления. Шаблоны.

В состав Laravel простой, но мощный шаблонизатор - **Blade**. Blade основан на концепции *наследования шаблонов* и *секциях документа*. Все шаблоны Blade должны иметь расширение.blade.php.

Создание шаблона Blade

```
<!-- Расположен в resources/views/layouts/master.blade.php -->
<html>
 <head>
 <title>App Name - @yield('title')</title>
 </head>
<body>
 @section('sidebar')
  Это - главная боковая панель.
 @show
  <div class="container">
   @yield('content')
  </div>
</body>
</html>
```

Laravel

* Представления. Шаблоны.

Использование шаблона Blade

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

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

@section('sidebar')
 @parent

 <p>Этот элемент будет добавлен к главной боковой панели.</p>
@endsecion

@section('content')
 <p>Это - содержимое страницы.</p>
@endsection
```

Шаблоны, которые расширяют другой Blade-шаблон с помощью **@extends**, просто перекрывают его секции. Старое (перекрытое) содержимое может быть выведено директивой **@parent**, что позволяет добавлять содержимое в такие секции, как боковая панель или «подвал».

Иногда может понадобиться указать значение по умолчанию для директивы **@yield**. Возможно передать его вторым аргументом:

```
@yield('section_name', 'Default Content')
```

Laravel

* Представления. Директивы Blade.

Вывод переменных

```
Hello, {{ $name }}.
The current UNIX timestamp is {{ time() }}.
```

Вывод переменных после проверки на существование

Иногда необходимо вывести значение переменной, но не известно, задано ли оно. Возможно использовать тернарный оператор:

```
{{ isset($name) ? $name : 'Default' }}
```

Вместо написания тернарного оператора, Blade позволяет использовать такое удобное сокращение:

```
{{ $name or 'Default' }}
```

Вывод необработанного текста в фигурных скобках

Если нужно вывести строку в фигурных скобках, можно отменить её обработку с помощью Blade, поставив перед текстом символ @:

```
@{{ Это не будет обработано с помощью Blade }}
```

Если не нужно экранировать данные(заменять HTML-сущности escape-последовательностями), следует использовать такой синтаксис:

```
Hello, {!! $name !!}.
```

Laravel

* Представления. Директивы Blade.

Директива If

```
@if (count($records) === 1)
  Здесь одна запись!
@elseif (count($records) > 1)
  Здесь много записей!
@else
  Здесь нет записей!
@endif

@unless (Auth::check())
  Вы не вошли в систему.
@endunless
```

Laravel

* Представления. Директивы Blade.

Циклы

```
@for ($i = 0; $i < 10; $i++)
  Текущее значение: {{ $i }}
@endfor
```

```
@foreach ($users as $user)
  <p>Это пользователь {{ $user->id }}</p>
@endforeach
```

```
@while (true)
  <p>Это будет длиться вечно.</p>
@endwhile
```

Laravel

Представления. Директивы Blade.

Включение подшаблонов

```
@include('view.name')
```

Также возможно передать массив данных во включаемый шаблон:

```
@include('view.name', ['some' => 'data'])
```

Перезапись секций

Для полной перезаписи можно использовать директиву **@overwrite**:

```
@extends('list.item.container')

@section('list.item.content')
  <p>Это - элемент типа {{ $item->type }}</p>
@overwrite
```

Комментарии

```
{{-- Этот комментарий не будет включён в сгенерированный HTML --}}
```

Laravel

* Основы работы с базами данных

Настройка

В Laravel процесс соединения с БД и выполнения запросов крайне прост. Настройки работы с БД хранятся в файле ***config/database.php***.

В этом файле можно указать все используемые соединения к БД, а также задать то, какое из них будет использоваться по умолчанию. Примеры настройки всех возможных видов подключений находятся в этом же файле.

На данный момент Laravel поддерживает 4 СУБД: MySQL, PostgreSQL, SQLite и SQL Server.

Laravel

* Основы работы с базами данных

Например:

```
'mysql' => array(
    'read' => array(
        'host' => '192.168.1.1',
    ),
    'write' => array(
        'host' => '196.168.1.2'
    ),
    'driver'    => 'mysql',
    'database'  => 'database_name',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

Laravel

* Основы работы с базами данных

Выполнение запросов

Как только соединение с базой данных настроено можно выполнять запросы, используя *фасад* DB.

Выполнение запроса SELECT

```
$results = DB::select('select * from users where id = ?', [1]);
```

Метод select() всегда возвращает массив.

Также можно выполнить запрос, используя привязку по имени:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Выполнение запроса INSERT

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Выполнение запроса UPDATE

```
DB::update('update users set votes = 100 where name = ?', ['John']);
```

Выполнение запроса DELETE

```
DB::delete('delete from users');
```

Laravel

* Основы работы с базами данных. **Eloquent ORM**

Система объектно-реляционного отображения (ORM) Eloquent — реализация шаблона ActiveRecord в Laravel для работы с базами данных. Каждая таблица имеет соответствующий класс-модель, который используется для работы с этой таблицей.

Основы использования

Для начала необходимо создать модель Eloquent. Модели обычно располагаются в папке app, но можно поместить их в любое место, в котором работает автозагрузчик в соответствии с файлом composer.json. Все модели Eloquent наследуют Illuminate\Database\Eloquent\Model.

Создание модели Eloquent

```
class User extends Eloquent {}
```

Генерировать модели Eloquent можно также с помощью команды make:model:

```
php artisan make:model User
```

* Основы работы с базами данных. **Eloquent ORM**

В примере выше не указано, какую таблицу Eloquent должен привязать к модели. Если это имя не указано явно, то будет использовано имя класса в нижнем регистре и во множественном числе. В данном случае Eloquent предположит, что модель User хранит свои данные в таблице users. Возможно указать произвольную таблицу, определив свойство $table в классе модели:

```
class User extends Eloquent {
  protected $table = 'my_users';
}
```

Eloquent также предполагает, что каждая таблица имеет первичный ключ с именем id. Возможно определить свойство $primaryKey для указания этого имени.

Laravel

* Основы работы с базами данных. **Eloquent ORM**

Получение всех моделей (записей)

```
$users = User::all();
```

Получение записи по первичному ключу

```
$user = User::find(1);
var_dump($user->name);
```

Получение модели по первичному ключу или возбуждение исключения

Если нужно сгенерировать исключение, когда определённая модель не была найдена – следует использовать метод firstOrFail():

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

Это позволит обработать исключение (например, занести его в журнал) и вывести необходимую страницу ошибки. Чтобы поймать исключение ModelNotFoundException, необходимо добавить обработчик в файл app/Exceptions/Handler.php.

Laravel

* Основы работы с базами данных. **Eloquent ORM**

Построение запросов в моделях Eloquent

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user){
    var_dump($user->name);
}
```

Агрегатные функции в Eloquent

Также доступны агрегатные функции, например:

```
$count = User::where('votes', '>', 100)->count();
```

Если не получается создать нужный запрос с помощью конструктора, то можно использовать метод whereRaw():

```
$users = User::whereRaw('age > ? and votes = 100', [25])->get();
```

* Основы работы с базами данных. **Eloquent ORM**

Разделение результата на блоки

Если нужно обработать много (тысячи) записей Eloquent, использование команды chunk позволит избежать переполнения оперативной памяти:

```
User::chunk(200, function($users){
    foreach ($users as $user){
        //
    }
});
```

Первый передаваемый в метод аргумент — число записей, получаемых в одном блоке. Передаваемая в качестве второго аргумента функция-замыкание будет вызываться для каждого блока, получаемого из БД.

Laravel

* Основы работы с базами данных. **Eloquent ORM**

Массовое заполнение

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

Необходимо определить в классе модели свойство $fillable или $guarded.

Указание доступных к заполнению атрибутов

Свойство $fillable указывает, какие поля должны быть доступны при массовом заполнении. Их можно указать на уровне класса или объекта.

```
class User extends Model {
    protected $fillable = ['first_name', 'last_name', 'email'];
}
```

В этом примере только три перечисленных поля будут доступны массовому заполнению.

* Основы работы с базами данных. **Eloquent ORM**

Указание охраняемых (*guarded*) атрибутов модели

Противоположность $fillable — свойство $guarded, которое содержит список запрещённых к заполнению полей:

```
class User extends Model {
    protected $guarded = ['id', 'password'];
}
```

Защита всех атрибутов от массового заполнения

В примере выше атрибуты id и password **не могут** быть присвоены через массовое заполнение. Все остальные атрибуты — могут. Вы также можете запретить все атрибуты для заполнения, используя символ *:

```
protected $guarded = ['*'];
```

* Основы работы с базами данных. **Eloquent ORM**

Вставка, обновление, удаление

Для создания новой записи в БД необходимо создать и проинициализировать экземпляр модели и вызвать метод save().

Сохранение новой модели

```
$user = new User;
$user->name = 'John';
$user->save();
```

Внимание: Модели Eloquent содержат автоинкрементные идентификаторы (*autoincrementing*). Однако если необходимо использовать собственные идентификаторы, нужно установить свойство $incrementing класса модели в значение false.

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

Laravel

* Основы работы с базами данных. **Eloquent ORM**

Создание модели

```
// Создание нового пользователя в БД...
$user = User::create(['name' => 'John']);

// Получение пользователя по свойствам, или его создание, если такого не суще
ствует...
$user = User::firstOrCreate(['name' => 'John']);

// Получение пользователя по свойствам, или создание нового экземпляра...
$user = User::firstOrNew(['name' => 'John']);
```

* Основы работы с базами данных. **Eloquent ORM**

Обновление полученной модели

Для обновления модели вам нужно получить её, изменить атрибут и вызвать метод save():

```
$user = User::find(1);
$user->email = 'john@foo.com';
$user->save();
```

Сохранение модели и её отношений

Иногда вам может быть нужно сохранить не только модель, но и все её отношения. Для этого используется метод push():

```
$user->push();
```

Вы также можете выполнять обновления в виде запросов к набору моделей:

```
$affectedRows = User::where('votes', '>', 100)->update(['status' => 2]);
```

* Основы работы с базами данных. **Eloquent ORM**

Удаление существующей модели

Для удаления модели вызовите метод delete() на её объекте:

```
$user = User::find(1);
$user->delete();
```

Удаление модели по ключу

```
User::destroy(1);
User::destroy([1, 2, 3]);
User::destroy(1, 2, 3);
```

Конечно, можно выполнять удаление на наборе моделей:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

* Основы работы с базами данных. **Eloquent ORM**

Мягкое удаление

При «мягком» удалении модели, она остаётся в базе данных, но устанавливается её поле deleted_at. Для включения мягких удалений на модели необходимо использовать SoftDeletes:

```
use Illuminate\Database\Eloquent\SoftDeletes;
class User extends Model {
    use SoftDeletes;
    protected $dates = ['deleted_at'];
}
```

Для отображения всех моделей, в том числе удалённых, используйте метод withTrashed():

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

Метод withTrashed может быть использован в отношениях:

```
$user->posts()->withTrashed()->get();
```

Если нужно получить **только** удалённые модели, вызовите метод onlyTrashed():

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

Для восстановления мягко удалённой модели в активное состояние используется метод restore():

```
$user->restore();
```

Основы работы с базами данных. **Eloquent ORM**

Если необходимо полностью удалить модель из БД, используется метод forceDelete():

```
$user->forceDelete();
```

Он также работает с отношениями:

```
$user->posts()->forceDelete();
```

Для того, чтобы узнать, удалена ли модель, можно использовать метод trashed():

```
if ($user->trashed())
{
  //
}
```

Поля времени

По умолчанию Eloquent автоматически поддерживает поля created_at и updated_at. Следует только добавьте эти *timestamp*-поля к таблице, и Eloquent позаботится об остальном. Если в этом нет необходимости:

```
class User extends Model {
    protected $table = 'users';
    public $timestamps = false;
}
```

Основы работы с базами данных. Eloquent ORM

Создание заготовки запроса

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

```
class User extends Model {
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }
    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Использование заготовки

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Основы работы с базами данных. Eloquent ORM

Динамические заготовки

Иногда вам может потребоваться определить заготовку, которая принимает параметры. Для этого просто добавьте эти параметры к методу заготовки:

```
class User extends Model {
    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }
}
```

А затем передайте их при вызове метода заготовки:

```
$users = User::ofType('member')->get();
```

Основы работы с базами данных. **Eloquent ORM**

Отношения

В большинстве случаев таблицы связаны с другими таблицами БД. Eloquent упрощает работу и управление такими отношениями. Laravel поддерживает многие типы связей:

Один к одному

Один ко многим

Многие ко многим

Ко многим через

Полиморфические связи

Полиморфические связи «многие ко многим»

Основы работы с базами данных. Eloquent ORM

Один к одному

Создание связи «один к одному»

Связь вида «один к одному» можно определить в Eloquent:

```
class User extends Model {
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

Первый параметр, передаваемый hasOne(), — имя связанной модели. Как только отношение установлено, вы можете получить к нему доступ через динамические свойства Eloquent:

```
$phone = User::find(1)->phone;
```

Сгенерированный SQL имеет такой вид:

```
select * from users where id = 1
select * from phones where user_id = 1
```

Основы работы с базами данных. **Eloquent ORM**

Eloquent считает, что поле в таблице называется по имени модели плюс _id. В данном случае предполагается, что это user_id. Если вы хотите перекрыть стандартное имя, передайте второй параметр методу hasOne(). Кроме того, возможно передать в метод третий аргумент, чтобы указать, какие локальные столбцы следует использовать для объединения:

```
return $this->hasOne('App\Phone', 'foreign_key');
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Создание обратного отношения

Для создания обратного отношения в модели Phone используется метод belongsTo() («принадлежит к»):

```
class Phone extends Model {
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```


В примере выше Eloquent будет искать поле user_id в таблице phones.

Основы работы с базами данных. **Eloquent ORM**

Если необходимо назвать внешний ключ по другому, следует передать имя вторым параметром в метод belongsTo():

```
class Phone extends Model {
    public function user()
    {
        return $this->belongsTo('App\User', 'local_key');
    }
}
```

Кроме того, третий параметр определяет имя связанного столбца в родительской таблице:

```
class Phone extends Model {
    public function user()
    {
        return $this->belongsTo('App\User', 'local_key', 'parent_key');
    }
}
```

Основы работы с базами данных. Eloquent ORM

Один ко многим

Примером отношения «один ко многим» является пост в блоге, который имеет комментарии:

```
class Post extends Model {
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Теперь мы можем получить все комментарии с помощью динамического свойства:

```
$comments = Post::find(1)->comments;
```

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

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

Второй параметр метода hasMany() может быть использован для перекрытия стандартного имени ключа. И как и для отношения «hasOne» также может быть указан локальный столбец:

```
return $this->hasMany('App\Comment', 'foreign_key');
return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Определение обратного отношения

Для определения обратного отношения так же используется метод belongsTo():

```
class Comment extends Model {
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Основы работы с базами данных. Eloquent ORM

Многие ко многим

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

Вы можете определить отношение «многие ко многим» через метод belongsToMany():

```
class User extends Model {
    public function roles() {
        return $this->belongsToMany('App\Role');
    }
}
```

Теперь мы можем получить роли через модель User:

```
$roles = User::find(1)->roles;
```

Основы работы с базами данных. **Eloquent ORM**

Вы можете передать второй параметр к методу belongsToMany() с указанием имени связующей (*pivot*) таблицы вместо стандартной:

```
return $this->belongsToMany('App\Role', 'user_roles');
```

Вы также можете перекрыть имена ключей по умолчанию:

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'foo_id');
```

Конечно, вы можете определить и обратное отношение на модели Role:

```
class Role extends Model {
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

Основы работы с базами данных. **Eloquent ORM**

Ко многим через

Связь «ко многим через» обеспечивает удобный короткий путь для доступа к удалённым отношениям через промежуточные. Например, модель Country может иметь много Post через модель User. Таблицы для этих отношений будут выглядеть так:

countries

- id - integer
- name - string

users

- id - integer
- country_id - integer
- name - string

posts

- id - integer
- user_id - integer
- title – string

Основы работы с базами данных. **Eloquent ORM**

Несмотря на то, что таблица posts не содержит столбца country_id, отношение «hasManyThrough» позволит нам получить доступ к posts через country с помощью $country->posts. Отношения:

```
class Country extends Model {
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

Если вы хотите указать ключи отношений вручную, вы можете передать их в качестве третьего и четвертого аргументов метода:

```
class Country extends Model {
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');
    }
}
```

Основы работы с базами данных. Eloquent ORM

Полиморфические отношения

Полиморфические отношения позволяют модели быть связанной с более, чем одной моделью. Например, может быть модель Photo, содержащая записи, принадлежащие к моделям Staff и Order. Мы можем создать такое отношение таким образом:

```
class Photo extends Model {
    public function imageable()
    {
        return $this->morphTo();
    }
}
class Staff extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
class Order extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
```

Чтение полиморфической связи

Теперь мы можем получить фотографии и для сотрудника, и для заказа:

```
$staff = Staff::find(1);
foreach ($staff->photos as $photo)
{
    //
}
```

При чтении связи на модели Photo:

```
$photo = Photo::find(1);
$imageable = $photo->imageable;
```

Отношение imageable модели Photo вернёт либо объект Staff, либо объект Order в зависимости от типа модели, которой принадлежит фотография.

Основы работы с базами данных. Eloquent ORM

Структура таблиц полиморфической связи

Рассмотрим структуру БД для полиморфического отношения

staff

- id - integer
- name - string

orders

- id - integer
- price - integer

photos

- id - integer
- path - string
- imageable_id – integer
- imageable_type – string

Главные поля, на которые нужно обратить внимание: **imageable_id** и **imageable_type** в таблице photos. Первое содержит ID владельца, в нашем случае — заказа или персонала, а второе — имя класса-модели владельца. Это позволяет ORM определить, какой класс модели должен быть возвращён при использовании отношения imageable.

Основы работы с базами данных. Eloquent ORM

Полиморфические связи многие ко многим

Структура таблиц полиморфической связи многие ко многим

В дополнение к традиционным полиморфическим связям вы можете также задать полиморфические связи многие ко многим. Например, модели блогов Post и Video могут разделять полиморфическую связь с моделью Tag. Во-первых, рассмотрим структуру таблиц:

posts(id – integer, name – string)

videos(id – integer, name – string)

tags(id – integer, name – string)

taggables(tag_id – integer, taggable_id – integer, taggable_type – string)

Далее, мы готовы к установке связи с моделью. Обе модели Post и Video будут иметь связь «morphToMany» через метод tags:

```
class Post extends Model {
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

Основы работы с базами данных. **Eloquent ORM**

Модель Tag может определить метод для каждого из своих отношений:

```
class Tag extends Model {
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

Основы работы с базами данных. Eloquent ORM

Запросы к отношениям

Проверка связей при выборке

При чтении отношений модели может быть необходимо ограничить результаты в зависимости от существования связи. Например, вы хотите получить все статьи в блоге, имеющие хотя бы один комментарий. Для этого можно использовать метод has():

```
$posts = Post::has('comments')->get();
```

Можно конструировать вложенные операторы has с помощью точечной нотации:

```
$posts = Post::has('comments.votes')->get();
```

Можно использовать методы whereHas и orWhereHas, чтобы поместить условия "where"в запросы has:

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');
})->get();
```

Основы работы с базами данных. Eloquent ORM

Динамические свойства

Eloquent позволяет читать отношения через динамические свойства. Eloquent автоматически определит используемую связь и вызовет get() для связей «один ко многим» и first() — для связей «один к одному». Эта связь будет доступна через динамическое свойство с тем же именем. К примеру, для следующей модели $phone:

```
class Phone extends Model {
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
$phone = Phone::find(1);
```

Вместо того, чтобы получить e-mail пользователя так:

```
echo $phone->user()->first()->email;
```

...вызов может быть сокращён до такого:

```
echo $phone->user->email;
```

Основы работы с базами данных. Eloquent ORM

Активная загрузка

Активная загрузка (*eager loading*) призвана устранить проблему запросов *N + 1*. Например, представьте, что у нас есть модель Book со связью к модели Author. Отношение определено как:

```
class Book extends Model {
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Теперь предположим, у нас есть цикл:

```
foreach (Book::all() as $book)
{
    echo $book->author->name;
}
```

Цикл выполнит один запрос для получения всех книг в таблице, а затем будет выполнять по одному запросу на каждую книгу для получения автора. Таким образом, если у нас 25 книг, то потребуется 26 запросов.

Основы работы с базами данных. Eloquent ORM

Возможно использовать активную загрузку для кардинального уменьшения числа запросов. Отношение будет активно загружено, если оно было указано при вызове метода with():

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

В цикле выше будут выполнены всего два запроса:

```
select * from books
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Разумное использование активной загрузки поможет сильно повысить производительность приложения.

Можно загрузить несколько отношений одновременно:

```
$books = Book::with('author', 'publisher')->get();
```

или вложенные отношения:

```
$books = Book::with('author.contacts')->get();
```

Основы работы с базами данных. **Eloquent ORM**

Ограничения активной загрузки

Иногда вам может быть нужно не только активно загрузить отношение, но также указать условие для его загрузки:

```
$users = User::with(['posts' => function($query)
{
    $query->where('title', 'like', '%первое%');

}])->get();
```

В этом примере мы загружаем сообщения пользователя, но только те, заголовок которых содержит подстроку «первое».

Функции-замыкания активной загрузки не ограничиваются только условиями. Вы также можете применить упорядочивание:

```
$users = User::with(['posts' => function($query)
{
    $query->orderBy('created_at', 'desc');

}])->get();
```

Основы работы с базами данных. **Eloquent ORM**

Ленивая активная загрузка

Можно активно загрузить связанные модели напрямую из уже созданного набора объектов моделей. Это может быть полезно при определении во время выполнения, требуется ли такая загрузка или нет, или в комбинации с кэшированием.

```
$books = Book::all();

$books->load('author', 'publisher');
```

Вы можете передать замыкание, чтобы задать ограничения для запроса:

```
$books->load(['author' => function($query)
{
    $query->orderBy('published_date', 'asc');
}]);
```

Основы работы с базами данных. **Eloquent ORM**

Связанные модели

Часто вам нужно будет добавить связанную модель. Например, вы можете создать новый комментарий к сообщению. Вместо явного указания значения для поля post_id вы можете вставить модель напрямую через её родителя — модели Post:

```
$comment = new Comment(['message' => 'A new comment.']);
$post = Post::find(1);
$comment = $post->comments()->save($comment);
```

В этом примере поле post_id вставленного комментария автоматически получит значение ID своей статьи.

Сохранить несколько связанных моделей можно так:

```
$comments = [
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another comment.']),
    new Comment(['message' => 'The latest comment.'])
];
$post = Post::find(1);
$post->comments()->saveMany($comments);
```

Основы работы с базами данных. **Eloquent ORM**

Связывание моделей (*belongs to*)

При обновлении связей belongsTo можно использовать метод associate(). Он установит внешний ключ на дочерней модели:

```
$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

Основы работы с базами данных. **Eloquent ORM**

Связывание моделей «многие ко многим»

```
$user = User::find(1);

$user->roles()->attach(1);
```

Вы также можете передать массив атрибутов, которые должны быть сохранены в связующей (*pivot*) таблице для этого отношения:

```
$user->roles()->attach(1, ['expires' => $expires]);
```

Конечно, существует противоположность attach() — detach():

```
$user->roles()->detach(1);
```

Оба метода attach() и detach() также принимают в качестве параметров массивы ID:

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

Основы работы с базами данных. Eloquent ORM

Использование sync() для привязки моделей «многие ко многим»

Вы также можете использовать метод sync() для привязки связанных моделей. Этот метод принимает массив ID, которые должны быть сохранены в связующей таблице. Когда операция завершится, переданные ID будут существовать в промежуточной таблице для данной модели:

```
$user->roles()->sync([1, 2, 3]);
```

Добавление данных для связующей таблицы при синхронизации

Вы также можете связать другие связующие таблицы с нужными ID:

```
$user->roles()->sync([1 => ['expires' => true]]);
```

Иногда вам может быть нужно создать новую связанную модель и добавить её одной командой. Для этого вы можете использовать метод save():

```
$role = new Role(['name' => 'Editor']);
User::find(1)->roles()->save($role);
```

В этом примере новая модель Role будет сохранена и привязана к модели User. Вы можете также передать массив атрибутов для помещения в связующую таблицу:

```
User::find(1)->roles()->save($role, ['expires' => $expires]);
```

Работа со связующими таблицами

Работа отношения многие ко многим требует наличия промежуточной таблицы. Например, предположим, что наш User имеет множество связанных объектов Role. После чтения отношения мы можем прочитать таблицу pivot на обеих моделях:

```
$user = User::find(1);
foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

Заметьте, что каждая модель Role автоматически получила атрибут pivot. Этот атрибут содержит модель, представляющую промежуточную таблицу, и она может быть использована как любая другая модель Eloquent.

Основы работы с базами данных. **Eloquent ORM**

По умолчанию, только ключи будут представлены в объекте pivot. Если ваша связующая таблица содержит другие поля, вы можете указать их при создании отношения:

```
return $this->belongsToMany('App\Role')->withPivot('foo', 'bar');
```

Теперь атрибуты foo и bar будут также доступны на объекте pivot модели Role.

Если вы хотите автоматически поддерживать поля created_at и updated_at актуальными, используйте метод withTimestamps() при создании отношения:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Удаление всех связующих записей

Для удаления всех записей в связующей таблице можно использовать метод detach():

```
User::find(1)->roles()->detach();
```

Заметьте, что эта операция не удаляет записи из таблицы roles, а только из связующей таблицы.

Laravel

Основы работы с базами данных. Eloquent ORM

Коллекции

Все методы Eloquent, возвращающие набор моделей — либо через get(), либо через отношения — возвращают объект-коллекцию. Этот объект реализует стандартный интерфейс PHP IteratorAggregate, что позволяет ему быть использованным в циклах как массив. Однако этот объект также имеет набор других полезных методов для работы с результатом запроса.

Проверка на существование ключа в коллекции

Например, мы можем выяснить, содержит ли результат запись с определённым первичным ключом, методом contains():

```
$roles = User::find(1)->roles;
if ($roles->contains(2))
{
    //
}
```

Коллекции также могут быть преобразованы в массив или строку JSON:

```
$roles = User::find(1)->roles->toArray();
$roles = User::find(1)->roles->toJson();
```

Проход по элементам коллекции

Коллекции Eloquent имеют несколько полезных методов для прохода и фильтрации содержащихся в них элементов:

```
$roles = $user->roles->each(function($role)
{
    //
});
```

Фильтрация элементов коллекции

При фильтрации коллекций передаваемая функция будет использована как функция обратного вызова для array_filter.

```
$users = $users->filter(function($user)
{
    return $user->isAdmin();
});
```

Основы работы с базами данных. **Eloquent ORM**

Применение функции к каждому объекту коллекции

```
$roles = User::find(1)->roles;

$roles->each(function($role)
{
    //
});
```

Сортировка коллекции по значению

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});

$roles = $roles->sortByDesc(function($role)
{
    return $role->created_at;
});
```

Основы работы с базами данных. **Eloquent ORM**

Сортировка коллекции по значению

```
$roles = $roles->sortBy('created_at');

$roles = $roles->sortByDesc('created_at');
```

Использование произвольного класса коллекции

Иногда вам может быть нужно получить собственный объект Collection со своими методами. Вы можете указать его при определении модели Eloquent, перекрыв метод newCollection():

```
class User extends Model {

    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

Миграции

Миграции —по сути система контроля версий для базы данных. Они позволяют команде изменять её структуру, в то же время оставаясь в курсе изменений других участников.

Создание миграций

Для создания новой миграции вы можете использовать Artisan-команду:

```
php artisan make:migration create_users_table
```

Миграция будет помещена в папку database/migrations и будет содержать метку времени, которая позволяет фреймворку определять порядок применения миграций.

Можно также использовать параметры --table и --create для указания имени таблицы и того факта, что миграция будет создавать новую таблицу (а не изменять существующую):

```
php artisan make:migration add_votes_to_users_table --table=users
```

```
php artisan make:migration create_users_table --create=users
```

Миграции

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateFlightsTable extends Migration
{
    /**
    * Run the migrations.
    * @return void
    */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }
    /**
    * Reverse the migrations.
    * @return void
    */
    public function down()
    {
        Schema::drop('flights');
    }
```

Миграции

Применение всех неприменённых миграций

php artisan migrate

Внимание: Если при применении миграции вы получаете ошибку «class not found» («класс не найден»), попробуйте выполнить команду **composer** dump-autoload.

Принудительные миграции в продакшне

Некоторые операции миграций разрушительны, они могут привести к потере данных. Для предотвращения случайного запуска этих команд на боевой БД перед их выполнением запрашивается подтверждение. Для принудительного запуска команд без подтверждения используйте ключ --force:

php artisan migrate –force

Миграции

Откат миграций

Отмена изменений последней миграции

php artisan migrate:rollback

Отмена изменений всех миграций

php artisan migrate:reset

Откат всех миграций и их повторное применение

php artisan migrate:refresh

Seed Загрузка начальных данных в БД

Кроме миграций, описанных выше, Laravel также включает в себя механизм наполнения вашей БД начальными данными (*seeding*) с помощью специальных классов. Все такие классы хранятся в database/seeds. Они могут иметь любое имя, но вам, вероятно, следует придерживаться какой-то логики в их именовании — например, UserTableSeeder и т.д. По умолчанию определён класс DatabaseSeeder. Из этого класса вы можете вызывать метод call() для подключения других классов с данными, что позволит вам контролировать порядок их выполнения.

Laravel

Загрузки данных

Пример класса для загрузки начальных данных

```
class DatabaseSeeder extends Seeder {
  public function run()
  {
    $this->call('UserTableSeeder');
    $this->command->info('Таблица пользователей загружена данными!');
  }
}

class UserTableSeeder extends Seeder {
  public function run()
  {
    DB::table('users')->delete();
    User::create(['email' => 'foo@bar.com']);
  }
}
```

Laravel

Загрузки данных

Для добавления данных в БД используйте Artisan-команду db:seed:

```
php artisan db:seed
```

По умолчанию команда **db:seed** вызывает класс DatabaseSeeder, который может быть использован для вызова других классов, заполняющих БД данными. Однако, вы можете использовать параметр **--class** для указания конкретного класса для вызова:

```
php artisan db:seed --class=UserTableSeeder
```

Вы также можете использовать для заполнения БД данными команду migrate:refresh, которая также откатит и заново применит все ваши миграции:

```
php artisan migrate:refresh --seed
```