Laravel 5

====

Документация на русском языке. Перевод официальной документации.

Перевод взят с гитхаба LaravelRUS

Дата компиляции: 16.05.2015

PDF-файл с русской документаций опубликован на сайте shtyrlyaev.ru.

- Первый взгляд
    - Установка
    - Контроллеры
    - Ошибки и логирование
- Обо всем поподробнее
    - Authentication
    - Cache
    - Core Extension
    - Events
    - Facades
    - Forms & HTML
    - Helpers
    - IoC Container
    - Localization
    - Mail
    - Package Development
    - Pagination
    - Queues
    - Security
    - Session
    - SSH
    - Templates
    - Unit Testing
    - Validation
- Работа с БД
    - Начала
    - Query Builder
- Artisan
    - Разработка Artisan-команд

Laravel 5.0

В Laravel 5.0 изменена дефолтная структура приложения. Теперь все приложение входит в стандарт автоматической загрузки PSR-4 целиком и является более подходящей основой для построения надежного приложения. Рассмотрим основные изменения:

Новая структура папок

Папки `app/models` больше нет. Теперь все ваши классы живут в папке app, и по умолчанию находятся в неймспейсе App. Название неймспейса хранится в файле `config/namespaces.php` и его можно изменить во всех ваших классах сразу артизан-командой `php artisan app:name`.

Контроллеры, middlewares (посредники, обработчики HTTP-запросов) и requests (новый вид классов в Laravel 5.0) сгруппированы в папке `app/Http` как классы, относящиеся к HTTP-слою вашего приложения. Вместо файла фильтров роутов теперь во фреймворке используются middlewares, которые находятся каждый в своём файле.

Новая папка `app/Providers` является заменой папки `app/start` в предыдущих версиях Laravel 4.x. В этой папке находятся сервис-провайдеры, которые осуществляют инициализацию приложения - регистрацию классов-обработчиков ошибок, настройку логирования, загрузку файла роутов (маршрутов) и т.п. И, конечно, ваши сервис-провайдеры тоже могут находиться там.

Файлы локализаций (lang) и файлы шаблонов (views) теперь находятся в папке `resources`.

Контракты

Все основные компоненты Laravel реализуют интерфейсы, размещенные в репозитории `illuminate/contracts`. У этого репозитория нет внешних зависимостей, это скелет фреймворка. Этот удобный корневой набор интерфейсов, который вы можете использовать в DI (dependency injection) своих классов, может служить альтернативой фасадам.

Документация по контрактам.

Кэширование роутов

Если ваше приложение использует много роутов, то для ускорения их обработки вы можете использовать artisan-команду `route:cache`. Эту команду можно применять на рабочем (продакшн) сервере после развёртывания (деплоя) приложения.

Middleware

В Laravel 5 появились так называемые middlewares, посредники, которые выполняют роль, которая раньше возлагалась на фильтры роутов. Фильтры роутов продолжают поддерживаться, но все встроенные фильтры HTTP-запросов, как то CSRF-фильтрация, проверка аутентификации, переехали в middlewares. Свои обработчики HTTP-запроса тоже лучше писать в виде middlewares.

Документация по middleware.

DI (dependency injection) в методах контроллеров

Основной способ передачи классов для использования в вашем контроллере - указать их (type-hint) в аргументах конструктора вашего контроллера. Так как Laravel создает контроллеры и другие классы фреймворка при помощи сервис-контейнера, он автоматически создает ожидаемые в аргументах конструктора классы и автоматически же подставляет их в вызов контроллера.

Теперь все вышеописанное работает не только для конструктора контроллера, но и для всех его методов.

```
public function createPost(Request $request, PostRepository $posts)
{
    //
}
```

Аутентификация из коробки

Laravel 5 содержит все необходимые миграции, контроллеры, модели и шаблоны для организации регистрации, аутентификации и смены пароля пользователя. Шаблоны находятся в `resources/views/auth`, валидация - `App\Services\Auth\Registrar`, контроллеры - в папке `app\Http\Controllers\Auth`. Теперь не нужно для каждого проекта писать код аутентификации, или копировать его из проекта в проект.

События-объекты

Теперь вы можете определить событите (event) как объект:

```
class PodcastWasPurchased {

    public $podcast;

    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

}
```

Запуск события осуществляется как обычно, только теперь вместо строки-имени события можно использовать экземпляр события-объекта:

```
Event::fire(new PodcastWasPurchased($podcast));
```

Конечно, ваш обработчик события в таком случае должен принимать объект вместо произвольной переменной `$data`:

```
class ReportPodcastPurchase {

    public function handle(PodcastWasPurchased $event)
    {
        //
    }

}
```

Документация по событиям.

Командная шина

В дополнение к задачам (job), помещаемым в очередь, которые вы использовали в Laravel 4, Laravel 5 предлагает концепцию команд, запускаемых через так называемую командную шину (command bus). Команды находятся в папке `app/Commands` и их тоже можно помещать в очередь (а можно и выполнять в текущем запросе). Вот пример команды:

```
class PurchasePodcast extends Command implements SelfHandling, ShouldBeQueued {

    use SerializesModels;

    protected $user, $podcast;

    /**
     * Создание объекта команды.
     *
     * @return void
     */
    public function __construct(User $user, Podcast $podcast)
    {
        $this->user = $user;
        $this->podcast = $podcast;
    }

    /**
     * Выполнение команды.
     *
     * @return void
     */
    public function handle()
    {
        // Handle the logic to purchase the podcast...

        event(new PodcastWasPurchased($this->user, $this->podcast));
    }

}
```

Базовый контроллер содержит трейт `DispatchesCommands` для выполнения команд:

```
$this->dispatch(new PurchasePodcastCommand($user, $podcast));
```

Команды - прекрасное средство для разбивки функционала вашего приложения на изолированные части и разгрузки ваших контроллеров.

Документация по командной шине

Реализация очереди в БД

Появился новый драйвер очереди - `database`. Теперь, если вы испытываете трудности с установкой дополнительного софта (Redis или Beanstalk) на ваш сервер, вы можете использовать таблицу MySQL для реализации очереди при помощи этого драйвера.

Встроенный шедулер (перидический запуск команд)

In the past, developers have generated a Cron entry for each console command they wished to schedule. However, this is a headache. Your console schedule is no longer in source control, and you must SSH into your server to add the Cron entries. Let's make our lives easier. The Laravel command scheduler allows you to fluently and expressively define your command schedule within Laravel itself, and only a single Cron entry is needed on your server.

Раньше нам приходилось каждый скрипт, который должен был запускаться с определённой частотой или в определённое время, руками заносить в Cron, соединяясь с сервером по SSH, а при переезде на другой сервер - не забывать копировать его руками. С Laravel 5 жизнь стала проще. Теперь вы можете настраивать периодический запуск внутри вашего кода и хранить в системе контроля версий.

Вы только посмотрите, как это красиво:

```
$schedule->command('artisan:command')->dailyAt('15:00');
```

Документация по периодическому запуску команд.

Tinker / Psysh

Команда `php artisan tinker` в Laravel 5 использует Psysh от Justin Hileman. Если вам нравился Boris в Laravel 4, вы полюбите и Psysh. Он лучше и работает под Windows!

DotEnv

В Laravel 5 больше нет папок для конфигов, специфичных для определённой среды выполнения. Все конфиги теперь хранятся в одной папке, а значения, специфичные среды выполнения хранятся в файле `.env`. Кроме того, название среды выполнения тоже задается в файле `.env` ! Больше никаких привязок к имени машины и флага `--env` в artisan-командах. Laravel 5 использует библиотеку DotEnv от Vance Lucas.

Документация по настройке среды приложения.

Laravel Elixir

Laravel Elixir от Jeffrey Way - это инструмент для сборки css и js вашего приложения. Если вы слышали о Grunt или Gulp, но использование их казалось вам слишком сложным - попробуйте Elixir. Elixir представляет собой удобную надстройку над Gulp. С помощью него вы легко сможете компилировать Less, Sass или CoffeeScript. Он даже может автоматически запускать тесты за вас !

Документация по Laravel Elixir.

Laravel Socialite

Laravel Socialite - пакет, совместимый с Laravel 5.0+ для аутентификации на сайте через OAuth-провайдеры. Поддерживаются Facebook, Twitter, Google и GitHub:

```
public function redirectForAuth()
{
    return Socialize::with('twitter')->redirect();
}

public function getUserFromProvider()
{
    $user = Socialize::with('twitter')->user();
}
```

Больше нет нужды подбирать работающие библиотеки. Все просто работает!

Документация по аутентификации через соцсети.

Облачная файловая система

Laravel 5 содержит Flysystem, пакет абстракции для работы с файловой системой, который поддерживает Amazon S3 и Rackspace Cloud Storage. Теперь вы можете одной строчкой в конфиге переключиться с хранения файлов на локальном диске на хранения файлов в облаке, так как функции работы с файлами не изменятся:

```
Storage::put('file.txt', 'contents');
```

Документация по Flysystem.

Form Requests

В Laravel 5.0 появились так называемые form requests. Это объект, который вместе с DI в методах контроллера, предлагает новый встроенный во фреймворк метод проверки и валидации пользовательского ввода.

Например, реквест-класс формы регистрации:

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

class RegisterRequest extends FormRequest {

    public function rules()
    {
        return [
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:8',
        ];
    }

    public function authorize()
    {
        return true;
    }

}
```

Далее вы вот так используете его в методе контроллера регистрации:

```
public function register(RegisterRequest $request)
{
    var_dump($request->input());
}
```

Когда сервис-контейнер видит, что в метод контроллера подключается класс типа `FormRequest`, запускается **автоматическая валидация** пользовательского ввода по правилам, заявленным в подключаемом классе. Если валидация не проходит, произойдет автоматический редирект с передачей ошибок валидации через сессии. **Валидация форм еще никогда не была такой простой**. Узнать больше про реквест-классы можно в соответствующей главе документации.

Валидация в контроллерах

Если создавать файл form request на каждый запрос слишком накладно для вас, вы можете валидировать запрос прямо в контроллере. В Laravel 5 это стало еще проще. Теперь в базовом контроллере есть трейт `ValidatesRequests`, который предоставляет метод `validate`:

```
public function createPost(Request $request)
{
    $this->validate($request, [
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
}
```

Вам не нужно контролировать результат валидации. Если валидация не удалась, фреймворк сам сделает редирект на предыдущую страницу, добавив в сессию сообщения об ошибках валидации и старый пользовательский ввод для отображения в форме. Или, если это был AJAX-запрос, вернёт соответствующий JSON.

Документация по валидации в контроллерах.

Новые генераторы

У фреймворка появились новые команды генерации классов, моделей и т.п. Смотрите `php artisan list` чтобы узнать подробности.

Кэш конфигов

You may now cache all of your configuration in a single file using the `config:cache` command.

Artisan-команда `config:cache` сливает конфиги в один файл для уменьшения количества операций чтения с диска.

Рекомендуется использовать эту команду на рабочих (продакшн) серверах после развертывания (деплоя) приложения.

VarDumper от Symfony

Хэлпер dd теперь использует прекрасный Symfony VarDumper, который раскрашивает дамп и позволяет схлопывать массивы. Посмотрите, как он работает:

```
dd([1, 2, 3]);
```

Сообщения об ошибках

Для активного развития, Laravel настоятельно рекомендует использовать только запросы на добавление изменений (pull requests), а не просто сообщения об ошибках (bug reports). Сообщение об ошибке так же может быть добавлено к pull request'у вместе с ошибками прохождения юнит-тестов.

Однако, если вы создали отчёт об ошибке, то он должен содержать заглавие и чёткое описание проблемы. Вы так же должны включить в него как можно более полную информацию и пример кода, которые помогут воспроизвести проблему. Основная цель отчёта об ошибке - упростить локализацию, воспроизведение проблемы и поиск её решения .

Также помните, что отчёты об ошибках создаются в надежде, что другие пользователи с такими же проблемами смогут принять участие в их решении вместе с вами. Но не ждите, что другие всё бросят и начнут исправлять вашу проблему. Отчёт об ошибке призван помочь вам и другим *начать совместную работу* над решением проблемы.

Исходный код Laravel находится на GitHub, список репозиториев для каждого проекта Laravel:

- Laravel Framework
- Laravel Application
- Laravel Documentation
- Laravel Cashier
- Laravel Envoy
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Website
- Laravel Art

Главное обсуждение разработки

Обсуждение, касающееся ошибок, новых функциональных возможностей и реализации уже существующих происходят на IRC канале `#laravel-dev` (Freenode). Тейлор Отвелл, главный разработчик Laravel, обычно присутствует на канале по будням с 8 утра до 5 вечера (UTC-06:00 или часовой пояс America/Chicago), иногда появляется спонтанно на канале в другое время.

IRC канал `#laravel-dev` открыт для всех. Мы рады всем, кто зашел на канал, не только чтобы принять участие, но и просто почитать обсуждения!

Какая ветка?

Все bug-fixes (исправления ошибок) должны отправляться в последюю стабильную версию ветки. Исправления ошибок **никогда** не должны отправляться в `master` ветку, если они только не исправляют функциональные возможности, которые есть только в последующем релизе.

Незначительные улучшения, **полностью обратно совместимые** с текущей версией Laravel, могут быть отправлены в последнюю стабильную ветку.

Серьёзные новые улучшения функциональных возможностей всегда должны отправляться в `master` ветку, которая содержит следующий релиз Laravel.

Если вы не уверены, относится ваше улучшение к незначительным или серьёзным, пожалуйста спросите у Тэйлора Отвелла на IRC канале `#laravel-dev` (Freenode).

Уязвимости в безопасности

Если вы обнаружили уязвимость в безопасности внутри Laravel, пожалуйста отправьте e-mail Тэйлору Отвеллу на почту taylorotwell@gmail.com. Все такие уязвимости будут незамедлительно рассмотрены.

Стиль написания кода

Laravel следует PSR-0 и PSR-1 стандартам. В дополнение к ним, необходимо так же следовать следующим стандартам:

- Декларация пространства имён (namespace) должна находиться на такой же строке, как и <?php.
- Открывающая фигурная скобка класса { должна находиться на такой же строке, как и название класса.
- Функции и управляющие конструкции должны использовать стиль Олмана:

    - Открывающая программная скобка должна располагаться на новой строке с тем же отступом, что и выражение, находящееся на предшествующей строке
    - Первое выражение внутри программных скобок должно располагаться на новой строке с отступом, увеличенным на 1 символ табуляции
    - Последующие выражения внутри программных скобок должны располагаться с тем же отступом, что и первое.
    - Закрывающая программная скобка должна располагается с отступом, равным отступу соответствующей ей открывающей программной скобке.
- Для отступов используется Tab, для выравнивания пробел.

Установка Composer

Laravel использует Composer для управления зависимостями. Поэтому прежде чем ставить Laravel вы должны установить Composer.

Установка Laravel

При помощи установщика Laravel

Используя Composer скачайте установщик Laravel.

```
composer global require "laravel/installer=~1.1"
```

Указав в качестве PATH директорию `~/.composer/vendor/bin`, станет возможным использование команды `laravel`.

После установки, простая команда `laravel new` создаст свеженькое Laravel приложение в директории, которую вы укажете. Например, `laravel new blog` создаст директорию `blog` и установит туда Laravel со всеми зависимостями. Этот метод установки намного быстрее, чем установка через Composer:

```
laravel new blog
```

При помощи Composer

Вы также можете установить Laravel используя команду Composer `create-project`:

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

Требования к серверу

У Laravel всего несколько требований к вашему серверу:

- PHP >= 5.4
- Mcrypt PHP Extension
- OpenSSL PHP Extension
- Mbstring PHP Extension

Начиная с PHP 5.5, в некоторых операционных системах может понадобиться ручная установка PHP JSON extension. В Ubuntu, например, это можно сделать при помощи `sudo apt-get install php5-json`.

Настройка

Первое, что вы должны сделать после установки Laravel - установить ключ шифрования сессий и кук. Это случайная строка из 32 символов, находится в файле `.env`, параметр 'APP_KEY'. Если вы устанавливали Laravel при помощи Composer, то ключ уже сгенерен. Вы можете сгенерить его вручную artisan-командой `key:generate`. **Если ключ шифрования отсутствует, ваши сессии, куки другая шифруемая информация не будет зашифрована надежным образом.**.

Laravel практически не требует другой начальной настройки - вы можете сразу начинать разработку. Однако может быть полезным изучить файл `config/app.php` - он содержит несколько настроек вроде `timezone` и `locale`, которые вам может потребоваться изменить в соответствии с нуждами вашего приложения.

Далее вы можете сконфигурить настройки среды выполнения.

> **Примечание:** Никогда не устанавливайте настройку `app.debug` в `true` на рабочем (продакшн) окружении .

Права на запись

Папки внутри `storage` должны быть доступны веб-серверу для записи. Если вы устанавливаете фреймворк на Linux или MacOS - открыть папки на запись можно командой `chmod -R 777 storage`

Красивые URL

Apache

Laravel поставляется вместе с файлом `public/.htaccess`, который настроен для обработки URL без указания

index.php. Если вы используете Apache в качестве веб-сервера обязательно включите модуль mod_rewrite.

Если стандартный .htaccess не работает для вашего Apache, попробуйте следующий:

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

Если вы используете в качестве веб-сервера Nginx, то используйте для ЧПУ следующую конструкцию:

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

Если вы используете Homestead, то вам ничего делать не нужно, там всё это уже настроено.

Вступление

Все файлы настроек Laravel хранятся в папке `config`. Опции хорошо описаны в комментариях, так что рекомендуем внимательно изучить эти файлы.

Действия после установки

Имя вашего приложения

После установки Laravel вы можете придумать имя своему приложению или оставить стандартное - App. Это имя будет фигурировать в качестве корневого элемента в пространстве имён классов, которые будут использоваться в вашем приложении. По умолчанию ваше приложение находится в папке app, имеет имя App и автозагружается при помощи Composer согласно стандарту PSR-4. Вы можете изменить его при помощи Artisan-команды `app:name`.

Например, чтобы изменить имя приложения на «Horsefly», выполните эту команду в корневом каталоге вашего приложения:

```
php artisan app:name Horsefly
```

Дальнейшее конфигурирование приложения

Laravel «из коробки» практически не требует дополнительного конфигурирования - вы можете сразу начать писать код. Максимум, вам может быть нужно изменить настройки доступа к базе данных в `config/database.php` и, возможно, изменить параметры `timezone` и `locale` в `config/app.php`.

Далее, вам нужно определиться с названием среды выполнения, в которой будет работать ваше приложение на данной машине. Например, когда вы разрабатываете приложение, вам нужно видеть подробный текст возникающих ошибок. По умолчанию фреймворк считает, что выполняется в среде `production` и в ней текст ошибки не выводится. Поэтому вы определяете, что на данной машине у вас среда выполнения `local` и в `config/local/app.php` ставите параметр debug в `true`.

> **Примечание** Никогда не ставьте `app.debug` в `true` в продакшне, т.е. на хостинге. Просто никогда.

Права на запись

Папки внутри `storage` должны быть доступны веб-серверу для записи. Если вы устанавливаете фреймворк на Linux или MacOs - открыть папки на запись можно командой `chmod -R 777 storage`

Получение и установка значений настроек

Для доступа к настройкам существует фасад `Config`:

```
$value = Config::get('app.timezone');

Config::set('app.timezone', 'America/Chicago');
```

Так же можно использовать функцию `config`:

```
$value = config('app.timezone');
```

Настройка среды выполнения

Часто необходимо иметь разные значения для разных настроек в зависимости от среды, в которой выполняется приложение. Например, вы можете захотеть использовать разные драйвера кэша на локальном и производственном (продакшн) серверах. Это легко достигается использованием настроек, зависящих от среды.

Для упрощения это задачи Laravel использует библиотеку DotEnv от Vance Lucas. В корне только что созданного приложения содержится файл `.env.example`. Если вы устанавливали Laravel через Composer, то этот файл был автоматически переименовал в `.env`, иначе вы должны сделать это самостоятельно.

Все переменные, описанные в этом файле, будут автоматически доступны вашему приложения в суперглобальной переменной `$_ENV`. Вы можете использовать функцию env для получения данных из этого массива внутри ваших файлов настроек. Если вы посмотрите в конфигурационные файлы, то увидите, что некоторые опции уже

устанавливаются с использованием этой функции! Не стесняйтесь изменять эти переменные под свои нужны, как на локальной машине, так и в продакшне.

Однако помните, что файл `.env` **не должен** попадать в систему контроля версий (добавьте его в `.gitignore`), так как каждый разработчик и каждый сервер могут требовать своих настроек среды выполнения, а так же это позволит спрятать ваши конфиденциальные данные от системы контроля версий.

Если вы работаете в команде, вы можете оставить файл `.env.example` с примерными значениями в составе приложения, что бы остальные члены команды понимали, какие переменные среды нужны для работы приложения.

Получение текущей среды

Вы можете получить текущую среду с помощью метода `environment` объекта `Application`:

```
$environment = $app->environment();
```

Вы также можете передать аргументы в этот метод чтобы проверить, совпадает ли среда с переданным значением:

```
if ($app->environment('local'))
{
    // Среда - local
}

if ($app->environment('local', 'staging'))
{
    // Среда - local ИЛИ staging
}
```

`$app` можно получить из сервис-контейнера по ключу `'Illuminate\Contracts\Foundation\Application'`. Если вы используете вышеприведенный код в сервис-провайдере, то вместо `$app` используйте `$this->app`.

Так же можно использовать функцию `app` и фасад `App`:

```
$environment = app()->environment();

$environment = App::environment();
```

Кэширование настроек

Что бы немного ускорить приложение, вы можете закэшировать настройки (объединить все файлы настроек в один), используя Artisan-команду `config:cache`.

Как правило, нужно добавить команду `config:cache` в ваш механизм развёртывания приложения (application deployment).

Режим обслуживания

Когда ваше приложение находится в режиме обслуживания (maintenance mode), специальный шаблон будет отображаться для всех запросов к вашему приложению. Это позволяет «отключать» ваше приложение во время обновления или обслуживания. Проверка на режим обслуживания уже включена в стандартный набор middleware вашего приложения. Если приложение находится в режиме обслуживания, то будет выброшено исключение `HttpException` с кодом 503.

Для включения этого режима просто выполните команду Artisan-а `down`:

```
php artisan down
```

Чтобы выйти из режима обслуживания выполните команду `up`:

```
php artisan up
```

Шаблон для режима обслуживания

Стандартный шаблон для режима обслуживания находится в файле `resources/views/errors/503.blade.php`.

Режим обслуживания и очереди

Пока ваше приложение находится в режиме обслуживания, очереди не будут обрабатываться. Работа очередей будет возобновлена, когда приложение выйдет из режима обслуживания.

Красивые URL

Apache

Laravel поставляется вместе с файлом `public/.htaccess`, который настроен для обработки URL без указания `index.php`. Если вы используете Apache в качестве веб-сервера обязательно включите модуль `mod_rewrite`.

Если стандартный `.htaccess` не работает для вашего Apache, попробуйте следующий:

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

Если вы используете в качестве веб-сервера Nginx, то используйте для ЧПУ следующую конструкцию:

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

Если вы используете Homestead, то вам ничего делать не нужно, там всё это уже настроено.

Введение

Laravel старается сделать восхитительной не только разработку на PHP, но и поднять на уровень выше вашу локальную среду разработки.

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

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

> **Примечание:** Если вы используете Windows, то вам надо разрешить в БИОСе аппаратную виртуализацию (VT-x).

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

Что внутри

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

Установка и настройка

Установка VirtualBox и Vagrant

Перед запуском среды Homestead, вы должны установить VirtualBox и Vagrant. Оба этих программных продукта имеют легкие в использовании установщики для всех популярных операционных систем.

Добавление бокса в Vagrant

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

```
vagrant box add laravel/homestead
```

Установка Homestead

При помощи Composer и PHP

На машине должен быть установлен PHP и Composer

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

```
composer global require "laravel/homestead=~2.0"
```

Проверьте, чтобы папка `~/.composer/vendor/bin` (`C:\Users\username\AppData\\Roaming\Composer\vendor\bin` в случае Windows) находилась у вас в PATH. Для проверки выполните в терминале команду `homestead`.

После установки инструмента, создайте конфигурационный файл `Homestead.yaml`:

```
homestead init
```

Файл `Homestead.yaml` будет создан в папке `~/.homestead` (или `C:\Users\username\.homestead` в случае Windows). Если вы используете Mac, Linux, или у вас в Windows стоит cygwin или msysgit (он ставится вместе с git), то вы можете редактировать его при помощи этой команды:

```
homestead edit
```

При помощи git

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

Склонируйте репозиторий с Homestead CLI в произвольную директорию:

```
git clone https://github.com/laravel/homestead.git Homestead
```

Выполните команду:

```
bash init.sh
```

Она создаст файл `Homestead.yaml` в папке `~/.homestead` (или `C:\Users\username\.homestead` в случае Windows)

SSH-ключи

Далее вам нужно отредактировать созданный `Homestead.yaml`. В этом файле вы можете указать путь к вашему публичному и приватному SSH-ключам, а также сконфигурировать совместно используемые локальной и виртуальной машиной папки.

Ни разу не использовали SSH ключи? Под Mac, Linux или Windows с установленным Git+msysgit вы можете создать пару ключей (приватный id*rsa и публичный id* rsa.pub), используя следующую команду:

```
ssh-keygen -t rsa -C "your@email.com"
```

В Windows в качестве альтернативы вы можете использовать PuTTYgen.

Укажите путь к публичному ключу в свойстве `authorize` файла `Homestead.yaml` , а путь к приватному - в свойстве `keys`.

Настройте общие папки

В свойстве `folders` в файле `Homestead.yaml` перечислены все локальные папки, доступ к которым вы хотите предоставить в среде Homestead. Файлы в этих папках будут синхронизироваться между локальной и виртуальной машиной. Синхронизация будет двусторонней. Настроить можно столько папок, сколько необходимо.

Настройте веб-сервер

Еще не знакомы с Nginx? Никаких проблем. Свойство `sites` позволяет легко связать домен и папку в среде Homestead. В файле `Homestead.yaml` имеется пример настройки одного сервера. Опять же, вы можете добавить столько сайтов, сколько вам нужно. Homestead может служить удобной виртуальной средой для нескольких проектов на Laravel.

Если вы хотите, чтобы ваш сайт работал под управлением HHVM, установите параметр `hhvm` в `true`:

```
sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public
      hhvm: true
```

Алиасы (aliases) Bash

Чтобы добавить произвольный алиас в Homestead, просто укажите его в файле `aliases` в папке `~/.homestead`.

Запуск Vagrant

После того как вы отредактировали файл `Homestead.yaml`, выполните в терминале команду `vagrant up` из каталога с установленным Homestead.

Vagrant запустит виртуальную машину, настроит синхронизацию папок и сконфигурирует веб-сервер Nginx согласно вашему конфигу.

Чтобы уничтожить виртуальную машину, выполните команду `vagrant destroy --force`.

Не забудьте добавить домены, которые вы будете использовать в Homestead, в файл `hosts`. Файл `hosts` будет перенаправлять ваши запросы к локальным доменам в среду Homestead. Под Mac и Linux это `/etc/hosts`. В Windows - `C:\Windows\System32\drivers\etc\hosts`. Строки, добавляемые вами в этот файл, будут выглядеть примерно так:

```
192.168.10.10  homestead.app
```

где 192.168.10.10 - это ip-адрес, прописанный в `Homestead.yaml`

После того, как вы добавите домен в файл `hosts`, вы получите доступ к сайту, развернутому в Homestead из вашего браузера.

```
http://homestead.app
```

Повседневное использование

Соединение По SSH

Чтобы подсоединиться к виртуальной машине Homestead по SSH, нужно выполнить в терминале команду `vagrant ssh` из каталога с установленным Homestead.

Вероятно, вам придется часто подключаться к вашей Homestead-машине, поэтому логичным будет создать «alias» на хост-машине:

```
alias vm="ssh vagrant@127.0.0.1 -p 2222"
```

После этого вы сможете подключаться к вашей Homestead-машине с помощью команды `vm` из любого каталога.

Коннект к базе данных

В Homestead установлены две СУБД - MySQL и Postgres. И там и там создана база данных `homestead`. В Laravel, дефолтных конфигах для среды выполнения `local`, уже указаны параметры для работы с этой базой данных.

Для соединения с СУБД Homestead вам нужно настроить клиент (Navicat, Sequel Pro, HeidiSQL и т.п.) на соединение с ip 127.0.0.1 и портом 33060 (MySQL) или 54320 (Postgres). Логин и пароль одинаковые для этих СУБД - `homestead` / `secret`

> **Примечание:** Эти нестандартные порты следует использовать только когда вы устанавливаете соединение из своей основной системы. В файлах конфигурации Laravel следует использовать порты по умолчанию 3306 и 5432, так как Laravel запускается *внутри* виртуальной машины.

Добавление сайтов

Через некоторое время вам, возможно, потребуется добавить домены новых сайтов в Homestead. Для этого есть два способа.

Во-первых, вы можете просто добавить сайты в файл `Homestead.yaml`, после чего выполнить `vagrant provision`.

Или же вы можете воспользоваться скриптом `serve`, доступным в среде Homestead. Для того, чтобы им воспользоваться, войдите по SSH в среду Homestead и выполните следующую команду:

```
serve domain.app /home/vagrant/Code/путь/к/директории/public
```

> **Замечание:** После запуска команды `serve` не забудьте добавить новый домен в файл `hosts` в вашей основной системе.

Порты

Список портов, которые перенаправляются из локальной машины в Homestead:

- **SSH:** 2222 → Перенаправление на порт 22
- **HTTP:** 8000 → Перенаправление на порт 80
- **MySQL:** 33060 → Перенаправление на порт 3306
- **Postgres:** 54320 → Перенаправление на порт 5432

Основы

Большинство роутов (маршруты, routes) вашего приложения будут определены в файле `app/Http/routes.php`, который загружается сервис-провайдером `App\Providers\RouteServiceProvider`. В Laravel простейший роут состоит из URI (урла, пути) и функции-замыкания.

Простейший GET-роут:

```
Route::get('/', function()
{
    return 'Hello World';
});
```

Простейший роуты различных типов HTTP-запросов:

```
Route::post('foo/bar', function()
{
    return 'Hello World';
});

Route::put('foo/bar', function()
{
    //
});

Route::delete('foo/bar', function()
{
    //
});
```

Регистрация роута для нескольких типов HTTP-запросов

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

Регистрация роута для любого типа HTTP-запроса:

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

Регистрация роута, всегда работающего через HTTPS:

```
Route::get('foo', array('https', function()
{
    return 'Must be over HTTPS';
}));
```

Вам часто может понадобиться сгенерировать URL к какому-либо роуту - для этого используется метод `URL::to`:

```
$url = url('foo');
```

Здесь 'foo' - это URI.

Защита от CSRF

Laravel provides an easy method of protecting your application from cross-site request forgeries. Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of the authenticated user.

Laravel предоставляет встроенное средство защиты от межсайтовых подделок запросов (cross-site request forgeries). Фреймворк автоматически генерит так называемый CSRF-токен для каждой активной сессии. Этот токен можно проверять при обработке запроса - действительно ли запрос послан с вашего сайта, а не с сайта-злоумышленника. Как правило проверяют только POST, PUT и DELETE запросы, так как они могут изменить состояние приложения (внести изменения в БД).

Вставка CSRF-токена в форму

```
<input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
```

То же самое, но с использованием шаблонизатора Blade:

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

You do not need to manually verify the CSRF token on POST, PUT, or DELETE requests. The `VerifyCsrfToken` HTTP middleware will verify token in the request input matches the token stored in the session.

Вам не нужно проверять вручную соответствие CSRF-токена сессионному в POST, PUT и DELETE запросах, middleware (посредник) `VerifyCsrfToken` делает это автоматически. Вдобавок к полю `_token`, проверяется еще и HTTP-заголовок `X-CSRF-TOKEN`, который часто используется в Javascript-фреймворках.

Подмена HTTP-метода

HTML-формы не поддерживают методы HTTP-запроса PUT или DELETE. Для того, чтобы отправить на сервер HTTP-запрос с этими методами, вам нужно добавить в форму скрытый input с именем `_method`.

Значение этого поля будет восприниматься фреймворком как тип HTTP-запроса. Например:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
</form>
```

Параметры роутов

В роутах можно указывать параметры, которые можно получить в виде аргументов функции-замыкания:

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

Необязательные параметры роута:

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

Необязательные параметры со значением по умолчанию:

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

Роуты с соответствием пути регулярному выражению:

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

Route::get('user/{id}', function($id)
{
    //
})
->where('id', '[0-9]+');
```

Конечно, при необходимости вы можете передать массив ограничений (constraints):

```
Route::get('user/{id}/{name}', function($id, $name)
{
    //
})
->where(['id' => '[0-9]+', 'name' => '[a-z]+'])
```

Определение глобальных паттернов

Вы можете определить соответствие параметра пути регулярному выражению глобально для всех роутов. Паттерны задаются в методе `before` сервис-провайдера `RouteServiceProvider`. Например, если у вас `{id}` в урлах это всегда числовое значение:

```
$router->pattern('id', '[0-9]+');
```

После того как паттерн определён, он применяется ко все роутам.

```
Route::get('user/{id}', function($id)
{
    // Здесь $id будет только числом
});
```

Доступ к значению параметров роута

Если вам нужно получить значение параметра вне роута, вы можете использовать метод `input()`:

```
if ($route->input('id') == 1)
{
    //
}
```

Вы можете обратиться к текущему роуту через объект `Illuminate\Http\Request`. Получить этот объект вы можете при помощи фасада `Request::` или подав его (type-hint) в аргументы конструктора нужного класса или в аргументы функции-замыкания:

```
use Illuminate\Http\Request;

Route::get('user/{id}', function(Request $request, $id)
{
    if ($request->route('id'))
    {
        //
    }
});
```

Именованные роуты

Присваивая имена роутам вы можете сделать обращение к ним (при генерации URL в шаблонах (views) или редиректах) более удобным. Вы можете задать имя роуту таким образом:

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

Также можно указать контроллер и его метод:

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

Теперь вы можете использовать имя маршрута при генерации URL или переадресации:

```
$url = route('profile');

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

Получить имя текущего выполняемого роута можно методом `currentRouteName`:

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

Группы роутов

Иногда вам может быть нужно применить фильтры к набору маршрутов. Вместо того, чтобы указывать их для каждого маршрута в отдельности вы можете сгруппировать маршруты:

```
Route::group(['middleware' => 'auth'], function()
{
    Route::get('/', function()
    {
        // К этому маршруту будет привязан фильтр auth.
    });

    Route::get('user/profile', function()
    {
        // К этому маршруту также будет привязан фильтр auth.
    });
});
```

Чтобы не писать полный неймспейс к каждому контроллеру, вы можете использовать параметр `namespace` в группе:

```
Route::group(['namespace' => 'Admin'], function()
{
    //
});
```

> **Примечание:** По умолчанию `RouteServiceProvider` включает все роуты внутрь неймспейса, определенного в его свойстве `$namespace`.

Поддоменные роуты

Роуты Laravel способны работать и с поддоменами:

Регистрация роута по поддомену:

```
Route::group(['domain' => '{account}.myapp.com'], function()
{

    Route::get('user/{id}', function($account, $id)
    {
        //
    });

});
```

Префикс пути

Группа роутов может быть зарегистрирована с одним префиксом без его явного указания с помощью ключа `prefix` в параметрах группы.

Добавление префикса к сгруппированным маршрутам:

```
Route::group(['prefix' => 'admin'], function()
{

    Route::get('user', function()
    {
        //
    });

});
```

Привязка моделей к роутам

Привязка моделей - удобный способ передачи экземпляров классов в ваш роут. Например, вместо передачи ID пользователя вы можете передать модель User, которая соответствует данному ID, целиком. Для начала используйте метод `route` в `RouteServiceProvider::boot` для указания класса, который должен быть внедрён.

Привязка параметра к модели

```
public function boot(Router $router)
{
    parent::boot($router);

    $router->model('user', 'App\User');
```

```
}
```

Затем зарегистрируйте роут, который принимает параметр `{user}`:

```
Route::get('profile/{user}', function(App\User $user)
{
    //
});
```

Из-за того, что мы ранее привязали параметр `{user}` к модели `App\User`, то её экземпляр будет передан в маршрут. Таким образом, к примеру, запрос `profile/1` передаст объект `User`, который соответствует ID 1, полученному из БД.

> **Внимание:** если переданный ID не соответствует строке в БД, будет брошено исключение (Exception) 404.

Если вы хотите задать свой собственный обработчик для события "не найдено", вы можете передать функцию-замыкание в метод `model`:

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

Иногда вам может быть нужно использовать собственный метод для получения модели перед её передачей в маршрут. В этом случае просто используйте метод `Route::bind`. Функция-замыкание принимает в качестве аргумента параметр из урла и должна возвратить экземпляр класса, который должен быть встроен в роут.

```
Route::bind('user', function($value)
{
    return User::where('name', $value)->first();
});
```

Ошибки 404

Есть два способа вызвать исключение 404 (Not Found) из маршрута. Первый - при помощи хэлпера `abort`:

```
abort(404);
```

Этот хэлпер бросает исключение `Symfony\Component\HttpFoundation\Exception\HttpException` с кодом ответа 404.

Второй - вы можете сами бросить исключение `Symfony\Component\HttpKernel\Exception\NotFoundHttpException`.

Смотрите также секцию errors документации.

Введение

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

> **Примечание:** Middlewares похожи на фильтры роутов в Laravel 4.

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

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

Создание middleware

Давайте для примера создадим middleware, который будет пропускать только те запросы, у которых параметр `age` будет больше чем 200, а всех остальных перенаправлять на `/home`.

Для создания middleware воспользуемся командой `make:middleware`:

```
php artisan make:middleware OldMiddleware
```

В папке `app/Http/Middleware` будет создан файл с классом `OldMiddleware`:

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

class OldMiddleware {

    /**
     * Run the request filter.
     *
     * @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);
    }

}
```

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

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

Регистрация middleware

Глобально

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

```
protected $middleware = [
    'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
    'Illuminate\Cookie\Middleware\EncryptCookies',
    'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
```

```
    'Illuminate\Session\Middleware\StartSession',
    'Illuminate\View\Middleware\ShareErrorsFromSession',
    'Illuminate\Foundation\Http\Middleware\VerifyCsrfToken',
];
```

Сопоставить с заданными роутами

Добавьте ваш middleware в свойство `routeMiddleware` класса `app/Http/Kernel.php`, назначив ему некоторое имя, например, `auth`, которое будет ключом массива:

```
protected $routeMiddleware = [
    'auth' => 'App\Http\Middleware\Authenticate',
    'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
    'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
];
```

Теперь вы можете назначить этот middleware роуту или группе:

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

Terminable Middleware

Иногда, middleware может понадобиться проделать некоторую работу после того как HTTP ответы были отправлены в браузер. Например, middleware "session", поставляемый с Laravel, записывает данные сессии в хранилище *после* отправки ответа браузеру. Для этого вы можете определить middleware как "terminable".

```
use Illuminate\Contracts\Routing\TerminableMiddleware;

class StartSession implements TerminableMiddleware {

    public function handle($request, $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Храним данные сессии...
    }

}
```

Как вы видите, в дополнение к определению метода `handle`, `TerminableMiddleware` определяет и `terminate` метод. Этот метод получает как запрос так и ответ. После того как вы определили terminable middleware, вы должны добавить его в список глобальных посредников в ваше ядро HTTP.

Введение

Вместо того, чтобы писать логику обработки запросов в файле `routes.php`, вы можете организовать её, используя классы Controller, которые позволяют группировать связанные обработчики запросов в отдельные классы.

Контроллеры обычно хранятся в папке `app/Http/Controllers`.

Простейшие контроллеры

Вот пример простейшего класса контроллера:

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

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }

}
```

Привязка «действия» (action) к определенному маршруту происходит так:

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

> **Примечание:** Все контроллеры должны наследоваться от базового класса контроллера.

Контроллеры и пространства имён

Очень важно отметить, что нам не пришлось указывать всё пространство имён контроллера, только часть названия класса, которая идёт после `App\Http\Controllers` - «корневого» пространства имён.

По умолчанию, класс `RouteServiceProvider` загружает все маршруты из файла `routes.php` в группу с «корневым» пространством имён.

Если вы решите наследовать или организовать ваши контроллеры используя пространства имён глубже в папке `App\Http\Controllers`, просто используйте название класса относительно корневого пространства имён `App\Http\Controllers`. Таким образом, если ваш полный класс `App\Http\Controllers\Photos\AdminController`, регистрация маршрута будет выглядеть так:

```
Route::get('foo', 'Photos\AdminController@method');
```

Именованные роуты с контроллерами

Как и в случае с обработчиками-замыканиями, вы можете присвоить имя этому роуту:

```
Route::get('foo', ['uses' => 'FooController@method', 'as' => 'name']);
```

Ссылки на «действия» контроллера

Если вы хотите создавать ссылки на «действие» контроллера, используя относительные имена имена классов, то сначала нужно зарегистрировать «корневое» пространство имён с контроллерами:

```
URL::setRootControllerNamespace('App\Http\Controllers');
```

Для непосредственного создания ссылки на «действие» контроллера, воспользуйтесь функцией-помощником `action`:

```
$url = action('FooController@method');
```

Получить имя «действия», которое выполняется в данном запросе, можно методом `currentRouteAction`:

```
$action = Route::currentRouteAction();
```

Использование посредников с контроллерами

Посредники могут быть привязаны к маршрутам следующим образом:

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

Так же, вы можете указывать их и в конструкторе самого контроллера:

```
class UserController extends Controller {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);
        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }

}
```

Единые контроллеры

Laravel позволяет вам легко создавать один маршрут для обработки всех действий контроллера. Для начала, зарегистрируйте маршрут методом `Route::controller`:

```
Route::controller('users', 'UserController');
```

Метод `controller` принимает два аргумента. Первый - корневой URI (путь), который обрабатывает данный контроллер, в то время как второй - имя класса самого контроллера. Далее просто добавьте методы в этот контроллер с префиксом в виде типа HTTP-запроса (HTTP verb), который они обрабатывают.

```
class UserController extends Controller {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

Методы `index` относятся к корневому URI (пути) контроллера, который, в нашем случае `users`.

Если имя действия вашего контроллера состоит из нескольких слов вы можете обратиться к нему по URI, используя синтаксис с дефисами (-). Например, данное действие в нашем классе `UserController` будет доступен по адресу `users/admin-profile`:

```
public function getAdminProfile() {}
```

Назначение имён

Если вы хотите задать имена для роутов, регистрируемых при помощи `Route::controller` , вы можете сделать это так:

```
Route::controller('users', 'UserController', [
    'anyLogin' => 'user.login',
]);
```

Где `anyLogin` - метод класса `UserController`.

RESTful ресурс-контроллеры

Ресурс-контроллеры упрощают построение RESTful контроллеров, работающих с ресурсами. Например, вы можете создать контроллер, обрабатывающий фотографии, хранимые вашим приложением. Вы можете быстро создать такой контроллер, используя Artisan-команду `make:controller`.

Для создания контроллера выполните следующую консольную команду:

```
php artisan make:controller PhotoController
```

Теперь мы можем зарегистрировать его как ресурс-контроллер:

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

Это единственное определение маршрута на самом деле описывает несколько маршрутов для обработки различных RESTful действий для ресурса `photo`. Сгенерированный контроллер уже имеет методы-заглушки для каждого из этих маршрутов с комментариями, которые напоминают о том, какие URI и типы запросов они обрабатывают.

Действия, обрабатываемые ресурс-контроллером:

Verb	Путь	Действие	Имя маршрута
GET	/photo	index	resource.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

Настройка маршрутов в ресурс-контроллерах

Иногда вам понадобится обрабатывать только часть всех возможных действий:

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

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

По умолчанию, все действия ресурс контроллеров имеют имя в формате «ресурс.действие». Однако, вы можете изменить эти имена, используя массив `names` в опциях:

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

Обработка наследуемых ресурс-контроллеров

Для того чтобы "наследовать" контроллеры ресурсов, используйте синтаксис с разделением точкой (`.`) при регистрации маршрутов:

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

Этот маршрут зарегистрирует "унаследованный" контроллер ресурсов, который может принимать URL такие как: `photos/{photos}/comments/{comments}`.

```
class PhotoCommentController extends Controller {

    /**
     * Show the specified photo comment.
     *
     * @param  int  $photoId
     * @param  int  $commentId
     * @return Response
     */
    public function show($photoId, $commentId)
    {
        //
    }

}
```

Добавление дополнительных маршрутов к ресурс контроллерам

Если вдруг необходимо добавить дополнительные маршруты к уже существующим маршрутам ресурс-контроллера, необходимо зарегистрировать эти маршруты **перед** вызовом метода `Route::resource`:

```
Route::get('photos/popular', 'PhotoController@method');

Route::resource('photos', 'PhotoController');
```

Внедрение зависимостей в контроллерах

Внедрение зависимостей в конструкторе

Сервис-контейнер используется для поиска и получения всех контроллеров. За счёт этого вы можете указать любые зависимости в качестве аргументов конструктора вашего контроллера, в том числе и любой контракт:

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

use Illuminate\Routing\Controller;
use App\Repositories\UserRepository;

class UserController extends Controller {

    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

}
```

Внедрение зависимостей в методах контроллеров

Точно так же можно указывать зависимости для любого метода контролера. Например, давайте укажем `Request` как зависимость метода `store`:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {
```

```
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }

}
```

Если метод контроллера так же ожидает какие-либо данные из маршрута, то можно просто указать их **после** всех зависимостей:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

> **Примечание:** Внедрение зависимостей в методы контроллера полностью дружат с механизмом связанных моделей. Сервис-контейнер определит, какие из аргументов должны быть связаны с моделями, а какие должны быть внедрены.

Кэширование маршрутов

Если ваше приложение использует только контроллеры для обработки маршрутов, вы можете использовать кэширование маршрутов, которое кардинально уменьшает время, требуемое на разбор и регистрацию всех маршрутов внутри приложения.

В некоторых случаях прирост скорости может достигнуть 100 раз! Для включения кэширования просто выполните Artisan-команду `route:cache`:

```
php artisan route:cache
```

Теперь сгенерированный кэш-файл маршрутов будет использоваться вместо файла `app/Http/routes.php`. Помните, что при добавлении новых или изменении существующих маршрутов необходимо пересоздавать кэш, поэтому лучше использовать кэширование маршрутов только при развёртывании приложения.

Для удаления кэша маршрутов без генерации нового используется Artisan-команда `route:clear`:

```
php artisan route:clear
```

Получение объекта HTTP-запроса

При помощи фасада

Фасад `Request` дает доступ к объекту HTTP-запроса:

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

Не забудьте использовать конструкцию `use Request;` в начале файла класса.

При помощи DI (dependency injection)

Можно получить объект HTTP-запроса при помощи DI (dependency injection, внедрение зависимости). Способ заключается в том, что в аргументы конструктора контроллера помещается (type-hint) объект, который нам нужен, и Laravel, когда создает контроллер, создает этот объект (см. сервис-контейнер) и подает на вход конструктору контроллера:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

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

        //
    }

}
```

Если ваш метод контроллера ожидает параметр из роута, укажите его после зависимостей:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

Входные данные

Получение входных данных

Объект `Illuminate\Http\Request` предоставляет доступ к входным данным, например, к переменным POST или PUT, полученным из формы. Вам не нужно указывать явно метод запроса, есть универсальный метод:

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

Получение переменной с дефолтным значением

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

Определение, содержится ли переменная в запросе

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

Получить все переменные запроса

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

Получить избранные переменные

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

$input = Request::except('credit_card'); // все, кроме этой
```

С массивами можно работать через нотацию с точкой:

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

Предыдущие входные данные

Часто нужно сохранять входные данные для следующего запроса. Например, это нужно для того, чтобы после ошибки пользователя в форме не заставлять его вводить всё заново, а заполнить правильные поля самим.

Сохранение запроса во flash-переменных сессии

Метод `flash` сохранит текущие входные данные в сессии, так, что они будут доступны в следующем запросе.

```
Request::flash();
```

Сохранение избранных переменных запроса

```
Request::flashOnly('username', 'email');

Request::flashExcept('password');
```

Редирект

Since you often will want to flash input in association with a redirect to the previous page, you may easily chain input flashing onto a redirect.

Чаще всего нам нужно сохранить данные и сделать редирект на урл с формой. В Laravel есть способ записать это просто и коротко:

```
return redirect('form')->withInput();

return redirect('form')->withInput(Request::except('password'));
```

Получение предыдущих данных

Чтобы получить сохранённые в сессии данные запроса, используйте метод `old`:

```
$username = Request::old('username');
```

Для использования в шаблонах можно использовать этот простой хэлпер:

```
{{ old('username') }}
```

Куки

Все куки, которые пишет Laravel, зашифрованы - это значит, что на клиенте они не могут быть изменены. Изменённую на клиенте куку фреймворк просто не сможет расшифровать и прочесть.

Получение значения куки

```
$value = Request::cookie('name');
```

Добавить новую куку к запросу

Хэлпер `cookie` создает объект `Symfony\Component\HttpFoundation\Cookie`. Полученный класс может быть добавлен к HTTP-ответу (`response`) методом `withCookie`:

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));
```

Создание вечной куки

"На самом деле нет". Время жизни "вечной" куки - 5 лет.

```
$response->withCookie(cookie()->forever('name', 'value'));
```

Файлы

Получение загруженного файла

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

Определение, загружался ли файл в запросе

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

Метод `file` возвращает экземпляр класса `Symfony\Component\HttpFoundation\File\UploadedFile`, который расширяет стандартный PHP-класс `SplFileInfo` и содержит все его методы.

Определение валидности загруженного файла

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

Перемещение загруженного файла

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

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

Другие методы работы с файлами

Полный список методов класса `Symfony\Component\HttpFoundation\File\UploadedFile` смотрите справке API.

Другая информация о запросе

The `Request` class provides many methods for examining the HTTP request for your application and extends the `Symfony\Component\HttpFoundation\Request` class. Here are some of the highlights.

Класс `Request` предоставляет множество методов позволяющих работать с HTTP-запросами в вашем приложении, является расширением класса `Symfony\Component\HttpFoundation\Request`. Вот некоторые из основых методов:

URI запроса

```
$uri = Request::path();
```

Метод запроса

```
$method = Request::method();
```

```
if (Request::isMethod('post'))
{
    //
}
```

Проверка на соответствие урла шаблону

```
if (Request::is('admin/*'))
{
    //
}
```

URL запроса

```
$url = Request::url();
```

Основы

HTTP-Response - это ответ фреймворка, который отдается клиенту (обычно это браузер), от которого пришел HTTP-запрос.

Наиболее простой способ создать HTTP-ответ - это возвратить строку в роуте или контроллере.

Response в виде возврата строки из роута:

```
Route::get('/', function()
{
    return 'Hello world';
});
```

Создание своих ответов

Однако чаще в контроллерах вы возвращаете объект `Illuminate\Http\Response` или шаблон. Возврат объекта `Response` позволяет изменить HTTP-код и заголовки ответа. Этот объект наследуется от `Symfony\Component\HttpFoundation\Response`, который предоставляет разнообразные методы для построения HTTP-ответа:

```
use Illuminate\Http\Response;

return (new Response($content, $status))
              ->header('Content-Type', $value);
```

Для удобства вы можете использовать хэлпер `response`:

```
return response($content, $status)
              ->header('Content-Type', $value);
```

> **Примечание:** Полный список методов `Response` можно увидеть в документации по API Laravel и документации по API Symfony.

Добавление контента в HTTP-ответ

Если вам нужно не просто изменить заголовки, но и вывести какой-то контент, вы можете указать имя шаблона при помощи метода `view()`:

```
return response()->view('hello')->header('Content-Type', $type);
```

Добавление куки

```
return response($content)->withCookie(cookie('name', 'value'));
```

Редиректы

Редирект - это объект класса `Illuminate\Http\RedirectResponse`, фактически это обычный HTTP-ответ без контента с установленным заголовком `Location`.

Возвращение редиректа

Есть несколько способов создать объект `RedirectResponse`. Самый простой - воспользоваться хэлпером `redirect`.

```
return redirect('user/login');
```

Возвращение редиректа с flash-данными в сессии

Редирект с flash-данными в сессии - типичная задача в случае, когда после POST-запроса надо перейти на страницу с формой и показать ошибки валидации. Записать flash-данные в сессию можно при помощи метода `with()`:

```
return redirect('user/login')->with('message', 'Login Failed');
```

Редирект на предыдущий URL

Для перехода назад к форме можно использовать также метод `back()`. Метод withInput() передаст данные, которые

пришли от этой формы, для того, чтобы отобразить их в форме и не заставлять пользователя снова вносить их.

```
return redirect()->back();

return redirect()->back()->withInput();
```

Редирект на именованный роут

Если использовать хэлпер `redirect()` без параметров, он вернет объект `Illuminate\Routing\Redirector`, у которого есть несколько интересных методов. При помощи них, например, вы можете сделать редирект на роут по его имени:

```
return redirect()->route('login');
```

Редирект на именованный роут с параметрами

Если ваш роут содержит параметры, то передать их вы можете так:

```
return redirect()->route('profile', [1]);
```

Если параметр роута - это ID некой модели, вы можете передать в аргументе экземпляр этой модели, Laravel возьмет оттуда ID сам:

```
return redirect()->route('profile', [$user]);
```

Редирект на именованный роут с именованными параметрами

```
// Если ваш роут с именем 'profile' имеет урл 'profile/{user}':

return redirect()->route('profile', ['user' => 1]);
```

Редирект на метод определённого контроллера

Вы можете также сделать редирект на экшн заданного контроллера:

```
return redirect()->action('App\Http\Controllers\HomeController@index');
```

> **Примечание:** Вам не нужно писать полный неймспейс контроллера, если вы задали его в `URL::setRootControllerNamespace`.

Редирект на метод определённого контроллера с параметрами

```
return redirect()->action('App\Http\Controllers\UserController@profile', [1]);
```

Редирект на метод определённого контроллера с именованными параметрами

```
return redirect()->action('App\Http\Controllers\UserController@profile', ['user' => 1]);
```

Особые HTTP-ответы

Если хэлпер `response()` вызывается без параметров, он возвращает имплементацию контракта `Illuminate\Contracts\Routing\ResponseFactory`, которая содержит несколько методов для генерации HTTP-ответа.

Отдача JSON

Метод `json` автоматически устанавливает заголовок `Content-Type` в `application/json`

```
return response()->json(['name' => 'Steve', 'state' => 'CA']);
```

Отдача JSONP

```
return response()->json(['name' => 'Steve', 'state' => 'CA'])
                 ->setCallback($request->input('callback'));
```

Отдача файла

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);
```

> **Примечание:** Классы Symfony HttpFoundation, которые занимаются функцией отдачи файла, требуют,

чтобы имя файла было в ASCII-формате.

Макросы

Вы можете оформить свой вариант HTTP-ответа в виде макроса, чтобы использовать его в других роутах или контроллерах в короткой форме.

HTTP-макросы определяются в методе `boot()` сервис-провайдера:

```
<?php namespace App\Providers;

use Response;
use Illuminate\Support\ServiceProvider;

class ResponseMacroServiceProvider extends ServiceProvider {

    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function($value)
        {
            return Response::make(strtoupper($value));
        });
    }

}
```

Используется макрос так:

```
return response()->caps('foo');
```

- Композеры

Основы использования

Views (представления, отображения, шаблоны) обычно содержат HTML-код вашего приложения и представляют собой удобный способ разделения бизнес-логики и логики отображения информации. Шаблоны хранятся в папке `resources/views`.

Простой шаблон выглядит примерно так:

```
<!-- resources/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

Из данного шаблона можно сформировать ответ в браузер примерно так:

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

Как вы можете видеть, первый аргумент хелпера `view` - имя файла шаблона в папке `resources/views`, а второй - данные, которые будут переданы в отображение.

Конечно, вы можете делать поддиректории в `resources/views`. Например, если ваш шаблон находится в файле `resources/views/admin/profile.php`, то вызов хэлпера будет выглядеть так:

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

Или так:

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

Передача данных в шаблон

Следующие примеры делают доступным в шаблоне переменную $name и присваивают ей значение 'Victoria':

```
// используя with()
$view = view('greeting')->with('name', 'Victoria');

// используя "магический" метод
$view = view('greeting')->withName('Victoria');

// при помощи второго параметра хэлпера
$view = view('greetings', ['name' => 'Victoria']);
```

Передача данных во все шаблоны

Иногда вам нужно передать данные во все шаблоны (например, это может быть залогиненый пользователь). Вы можете сделать это несколькими путями: при помощи хэлпера view(), используя `Illuminate\Contracts\View\Factory` contract или при помощи общего шаблона (wildcard) композера.

При помощи хэлпера это можно сделать вот так:

```
view()->share('data', [1, 2, 3]);
```

Или, если вы хотите использовать фасад, как в Laravel 4.x:

```
View::share('data', [1, 2, 3]);
```

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

> **Примечание:** Когда хэлпер `view()` вызывается без аргументов, он возвращает имплементацию контракта `Illuminate\Contracts\View\Factory`

Определение наличия шаблона

Чтобы определить, есть ли заданный файл шаблона на диске, используйте метод `exist()`

```
if (view()->exists('emails.customer'))
{
    //
}
```

Получение шаблона по полному пути

Вы можете взять файл шаблона по полному пути до него в файловой системе:

```
return view()->file($pathToFile, $data);
```

Композеры

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

Определение композера

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

```
<?php namespace App\Providers;

use View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider {

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
        // Если композер реализуется при помощи класса:
        View::composer('profile', 'App\Http\ViewComposers\ProfileComposer');

        // Если композер реализуется в функции-замыкании:
        View::composer('dashboard', function()
        {

        });
    }

    /**
        * Register
        *
        * @return void
        */
        public function register()
        {
            //
        }

}
```

> **Примечание:** В Laravel нет папки, в которой должны находиться классы композеров, вы можете создать её сами там, где вам будет удобно. Например, это может быть `App\Http\Composers`.

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

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

use Illuminate\Contracts\View\View;
use Illuminate\Users\Repository as UserRepository;

class ProfileComposer {
```

```
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Зависимости разрешаются автоматически службой контейнера...
        $this->users = $users;
    }

    /**
     * Bind data to the view.
     *
     * @param  View  $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }

}
```

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

> **Примечание:** Все композеры берутся из сервис-контейнера, поэтому вы можете указать необходимые зависимости в конструкторе композера - они будут автоматически поданы ему.

Wildcards имен шаблонов композеров

В названии шаблонов можно использовать wildcards (паттерны). Например, вот так можно назначить композер для всех шаблонов:

```
View::composer('*', function()
{
    //
});
```

Назначение композера для нескольких шаблонов

Вместо одного имени шаблона вы можете использовать массив имен:

```
View::composer(['profile', 'dashboard'], 'App\Http\ViewComposers\MyViewComposer');
```

Регистрация нескольких композеров

Вы можете использовать метод `composers` чтобы зарегистрировать несколько композеров одновременно:

```
View::composers([
    'App\Http\ViewComposers\AdminComposer' => ['admin.index', 'admin.profile'],
    'App\Http\ViewComposers\UserComposer' => 'user',
    'App\Http\ViewComposers\ProductComposer' => 'product'
]);
```

Создатель (view creators)

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

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

Введение

Service providers (сервис-провайдеры, дословно - «поставщики услуг») занимают центральное место в архитектуре Laravel. Они предназначены для первоначальной загрузки (bootstraping) приложения. Ваше приложение, а также сервисы самого фреймворка загружаются через сервис-провайдеры.

Что конкретно означает термин «первоначальная загрузка» или «bootsraping»? Главным образом это **регистрация** некоторых вещей - таких как биндинги в IoC-контейнер (фасадов и т.д.), слушателей событий (event listeners), фильтров роутов (route filters) и самих роутов (routes). Сервис-провайдеры - центральное место для конфигурирования вашего приложения.

Если вы откроете файл `config/app.php`, вы увидите массив `providers`. В нем перечислены все классы сервис-провайдеров, которые загружаются при старте вашего приложения (конечно, кроме тех, которые являются «отложенными» (deferred), т.е. загружаются по требованию другого сервис-провайдера).

Можно и нужно создавать свои собственные сервис-провайдеры для загрузки и настройки различных частей своего приложения.

Использование провайдеров

Сервис-провайдеры должны расширять (extends) класс `Illuminate\Support\ServiceProvider`. Это абстрактный класс, который требует, чтобы в наследуемом классе был метод `register()`. В методе `register()` вы можете **только** регистрировать свои классы (bindings) в сервис-контейнере. Слушателей событий (event listeners), роуты и фильтры роутов там регистрировать **нельзя**.

С помощью Artisan можно легко создать нового провайдера, используя команду `make:provider`:

```
php artisan make:provider RiakServiceProvider
```

Метод register()

Вот так может выглядеть простейший сервис-провайдер:

```
<?php namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider {

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton('Riak\Contracts\Connection', function($app)
        {
            return new Connection($app['config']['riak']);
        });
    }

}
```

В `register()` мы регистрируем (bind) как singleton (т.е. класс не будет переинициализироваться после вызова из контейнера) в сервис-контейнере класс работы с базой данных Riak. Если для вас этот код выглядит абракадаброй, не беспокойтесь, работу сервис-контейнера мы рассмотрим позже.

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

Метод boot()

Когда вызвались методы `register()` всех сервис-провайдеров приложения, вызывается метод `boot()` сервис-провайдеров. Там уже можно использовать весь существующий функционал классов фреймворка и вашего

приложения - регистрировать слушателей событий, подключать роуты и т.п.

```
<?php namespace App\Providers;

use Event;
use Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider {

    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot(Dispatcher $events)
    {
        Event::listen('SomeEvent', 'SomeEventHandler');
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

}
```

Обратите внимание, что сервис-контейнер, вызывая метод `boot()`, сам внедрит те зависимости, которые вы зададите, в частности, Dispatcher.

```
use Illuminate\Contracts\Events\Dispatcher;

public function boot(Dispatcher $events)
{
    $events->listen('SomeEvent', 'SomeEventHandler');
}
```

Регистрация провайдеров

Все сервис-провайдеры регистрируются в файле `config/app.php` путем добавления в массив `providers`. Все сервис-провайдеры фреймворка находятся там.

Чтобы зарегистрировать свой сервис-провайдер, добавьте название класса в этот массив:

```
'providers' => [
    // другие сервис-провайдеры

    'App\Providers\AppServiceProvider',
],
```

Отложенные провайдеры

Если ваш провайдер **только** регистрирует (bind) классы в сервис-контейнере, то вы можете отложить вызов его метода `register()` до момента, когда эти классы будут затребованы из сервис-контейнера. Это позволит не дергать файловую систему каждый запрос в попытках загрузить файл с нужным классом с диска.

Для того, чтобы сделать сервис-провайдер отложенным, установите свойство `defer` в `true` и определить метод `provides()`, чтобы фреймворк знал, какие классы биндятся (регистрируются в сервис-контейнере, «связываются») в вашем провайдере.

```
<?php namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider {

    /**
```

```
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton('Riak\Contracts\Connection', function($app)
        {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Get the services provided by the provider.
     *
     * @return array
     */
    public function provides()
    {
        return ['Riak\Contracts\Connection'];
    }

}
```

Laravel в процессе запуска собирает данные об отложенных сервис-провайдерах и классах, которые ими регистрируются, и когда в процессе работы приложению понадобится класс `Riak\Contracts\Connection`, он вызовет метод `register()` сервис провайдера `RiakServiceProvider`.

Введение

Service Container (сервис-контейнер, ранее IoC-контейнер) - это мощное средство для управлением зависимостями классов. В современном мире веб-разработки есть такой модный термин - Dependency Injection, «внедрение зависимостей», он означает внедрение неких классов в создаваемый класс через конструктор или метод-сеттер. Создаваемый класс использует эти классы в своей работе. Сервис-контейнер реализует как раз этот функционал.

Несколько упрощая, можно сказать так: когда фреймворку нужно создать класс, он применяет не конструкцию `new SomeClass(new SomeService())`, а `App::make('SomeClass')`, предварительно зарегистрировав функцию, которая создает класс `SomeClass` и все классы, которые `SomeClass` принимает в качестве аргументов конструктора.

Вот простой пример:

```
<?php namespace App\Handlers\Commands;

use App\Commands\PurchasePodcast;
use Illuminate\Contracts\Mail\Mailer;

class PurchasePodcastHandler {

    /**
     * The mailer implementation.
     */
    protected $mailer;

    /**
     * Create a new instance.
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    /**
     * Purchase a podcast.
     *
     * @param  PurchasePodcastCommand  $command
     * @return void
     */
    public function handle(PurchasePodcastCommand $command)
    {
        //
    }

}
```

В этом примере нам нужно в обработчике `PurchasePodcast` написать письмо пользователю для подтверждения покупки. Так как мы хотим соблюдать первый принцип SOLID - «Принцип разделения ответственности», мы не пишем в нём код общения с SMTP-сервером и т.п., а встраиваем, внедряем (inject) в него класс отправки мейлов. Преимущество такого подхода - не изменяя код класса `PurchasePodcast` мы можем легко сменить способ отправки почты, например, с сервиса MailChimp на Mailjet или другой, а для тестирования можем использовать класс-заглушку.

Сервис-контейнер - очень важная вещь, без него невозможно построить действительно большое приложение Laravel. Также глубокое понимание его работы необходимо, если вы хотите изменять код ядра Laravel и предлагать новые фичи.

Использование

Связывание (Binding, регистрация)

Так как практически все биндинги, т.е. соответствие строкового ключа реальному объекту в контейнере, в вашем

приложении будут регистрироваться в методе `register()` сервис-провайдеров, все нижеследующие примеры даны для этого контекста. Если вы хотите использовать контейнер в другом месте своего приложения, вы можете внедрить в свой класс `Illuminate\Contracts\Container\Container`. Так же для доступа к контейнеру можно использовать фасад App. (TODO дополнить примерами)

Регистрация обычного класса

Внутри сервис-провайдера экземпляр контейнера находится в `$this->app`.

Зарегистрировать (bind, связать) класс можно двумя путями - при помощи коллбэк-функции или привязки интерфейса к реализации.

Рассмотрим первый способ. Коллбэк регистрируется в сервис-контейнере под неким строковым ключом (в данном случае FooBar) - обычно для этого используют название класса, который будет возвращаться этим коллбэком:

```
$this->app->bind('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

Когда из контейнера будет запрошен объект по ключу FooBar, контейнер создаст объект класса FooBar, в конструктор которого в качестве аргумента добавит объект из контейнера с ключом SomethingElse.

Регистрация класса-синглтона

Иногда вам нужно, чтобы объект создавался один раз, а все остальные разы, когда вы запрашиваете его, вам возвращался тот же созданный экземпляр. В этом случае вместо bind используйте `singleton`:

```
$this->app->singleton('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

Добавление существующего экземпляра класса в контейнер

Вы можете добавить в контейнер существующий экземпляр класса:

```
$fooBar = new FooBar(new SomethingElse);

$this->app->instance('FooBar', $fooBar);
```

Получение из контейнера

Есть несколько способов получить (resolve) содержимое контейнера. Во-первых, вы можете использовать метод `make()`:

```
$fooBar = $this->app->make('FooBar');
```

Во-вторых, вы можете обратиться к контейнеру как к массиву:

```
$fooBar = $this->app['FooBar'];
```

И, наконец, в-третьих (и в главных) вы можете явно указать тип аргумента в конструкторе класса и фреймворк сам возьмёт его из контейнера (в примере ниже это `UserInterface`):

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

use Illuminate\Routing\Controller;
use App\Users\Repository as UserRepository;

class UserController extends Controller {

    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
```

```
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the user with the given ID.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        //
    }

}
```

Связывание интерфейса с реализацией

Инъекции зависимостей

Особенно интересная и мощная возможность сервис-контейнера - связывать интерфейсы с различными их реализациями. Например, наше приложение использует Pusher для отправки и приема push-сообщений. Если мы используем Pusher PHP SDK, мы должны внедрить экземпляр класса `PusherClient` в наш класс:

```
<?php namespace App\Handlers\Commands;

use App\Commands\CreateOrder;
use Pusher\Client as PusherClient;

class CreateOrderHandler {

    /**
     * The Pusher SDK client instance.
     */
    protected $pusher;

    /**
     * Create a new order handler instance.
     *
     * @param  PusherClient  $pusher
     * @return void
     */
    public function __construct(PusherClient $pusher)
    {
        $this->pusher = $pusher;
    }

    /**
     * Execute the given command.
     *
     * @param  CreateOrder  $command
     * @return void
     */
    public function execute(CreateOrder $command)
    {
        //
    }

}
```

Все бы ничего, но наш код становится завязанным на конкретный сервис - Pusher. Если в дальнейшем мы заходим его сменить, или просто Pusher сменит названия методов в своем SDK, мы будем вынуждены менять код в нашем классе `CreateOrderHandler`.

От класса к интерфейсу

Для того, чтобы «изолировать» класс `CreateOrderHandler` от постоянно меняющегося внешнего мира, определим некий постоянный интерфейс, с реализациями которого наш класс будет теперь работать.

```
<?php namespace App\Contracts;
```

```
interface EventPusher {

    /**
     * Push a new event to all clients.
     *
     * @param  string  $event
     * @param  array  $data
     * @return void
     */
    public function push($event, array $data);

}
```

Когда мы создадим реализацию (implementation) этого интерфейса, `PusherEventPusher`, мы можем связать её с интерфейсом в методе `register()` сервис-провайдера:

```
$this->app->bind('App\Contracts\EventPusher', 'App\Services\PusherEventPusher');
```

Здесь мы говорим фреймворку, что когда из контейнера будет запрошен `EventPusher`, вместо него отдавать реализацию этого интерфейса, `PusherEventPusher`. Теперь мы можем переписать наш конструктор класса `CreateOrderHandler` следующим образом:

```
/**
 * Create a new order handler instance.
 *
 * @param  EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
    $this->pusher = $pusher;
}
```

Теперь, с какой бы реализацией работы реалтаймовых сообщений мы бы ни работали, изменять код в `CreateOrderHandler` нам не потребуется.

Контекстное связывание

Иногда у вас может быть несколько реализаций одного интерфейса и вы хотите внедрять их каждый в свой класс. Например, когда делается новый заказ, вам нужно отправлять сообщение в PubNub вместо Pusher. Вы можете сделать это следующим образом:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
          ->needs('App\Contracts\EventPusher')
          ->give('App\Services\PubNubEventPusher');
```

Тэгирование

Иногда вам может потребоваться ресолвить реализации в определенной категории. Например, вы пишете сборщик отчётов, который принимает на вход массив различных реализаций интерфейса `Report`. Вы можете протэгировать их следующим образом:

```
$this->app->bind('SpeedReport', function()
{
    //
});

$this->app->bind('MemoryReport', function()
{
    //
});

$this->app->tag(['SpeedReport', 'MemoryReport'], 'reports');
```

Теперь вы можете получить их все сразу по тэгу:

```
$this->app->bind('ReportAggregator', function($app)
{
    return new ReportAggregator($app->tagged('reports'));
});
```

Использование на практике

Laravel предлагает несколько возможностей использования сервис-контейнера для повышения гибкости и тестируемости вашего кода. Один их характерных примеров - реализация Dependency Injection в контроллерах. Laravel регистрирует все контроллеры в сервис-контейнере и поэтому при получении (resolve) класса контроллера из контейнера, автоматически получаются все зависимости, указанные в аргументах конструктора и других методов контроллера.

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

use Illuminate\Routing\Controller;
use App\Repositories\OrderRepository;

class OrdersController extends Controller {

    /**
     * The order repository instance.
     */
    protected $orders;

    /**
     * Create a controller instance.
     *
     * @param  OrderRepository  $orders
     * @return void
     */
    public function __construct(OrderRepository $orders)
    {
        $this->orders = $orders;
    }

    /**
     * Show all of the orders.
     *
     * @return Response
     */
    public function index()
    {
        $all = $this->orders->all();

        return view('orders', ['all' => $all]);
    }

}
```

В этом примере `OrderRepository` будет автоматически создан и подан как аргумент конструктору. Во время тестирования вы можете связать ключ `'OrderRepository'` с классом-заглушкой и абстрагироваться от слоя базы данных, протестировав только функционал самого класса `OrdersController`.

Другие примеры

Разумеется, контроллеры не единственные классы, которые фреймворк берет из сервис-контейнера. Вы можете использовать этот же принцип в обработчиках маршрутов, событий, очередей и т.д. Примеры использования сервис-контейнера приведены в соответствующих разделах документации.

События контейнера

Регистрация события на извлечение объекта из контейнера

Сервис-контейнер запускает событие каждый раз, когда объект извлекается из контейнера. Можно ловить все события, можно только те, которые привязаны к конкретному ключу.

```
$this->app->resolvingAny(function($object, $app)
{
    //
});

$this->app->resolving('FooBar', function($fooBar, $app)
{
    //
});
```

Объект, получаемый из контейнера, передается в функцию-коллбэк.

Введение

Контракты в Laravel - это набор классов-интерфейсов, определяющий некий функционал ядра фреймворка. Например, контракт `Queue` определяет методы работы с очередями, `Mailer` - методы для отправки мейлов.

Каждый контракт имеет свою реализацию (implementation) во фреймворке. Например, есть реализация `Queue` с различными драйверами очередей и реализация `Mailer` с использованием SwiftMailer.

Для удобства, контракты находятся в отдельном репозитории на гитхабе - https://github.com/illuminate/contracts.

Зачем нужны контракты?

Вы можете спросить - а для чего нужны контракты? Зачем вообще нужны интерфейсы?

Ответ - слабая связность и упрощение кода.

Слабая связность

Сначала давайте рассмотрим пример кода с сильной связностью.

```
<?php namespace App\Orders;

class Repository {

    /**
     * The cache.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param  \Package\Cache\Memcached  $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
        $this->cache = $cache;
    }

    /**
     * Retrieve an Order by ID.
     *
     * @param  int  $id
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id))
        {
            //
        }
    }

}
```

Код этого класса тесно связан с реализацией кэширования `\SomePackage\Cache\Memcached`. Мы зависим и от способа кэширования (`memcached`), и от API данной библиотеки. Если мы хотим сменить кэширование с `memcached` на `redis`, нам придется вносить изменения в код класса `Repository`.

Чтобы такого не происходило, класс `Repository` не должен задумываться, кто именно предоставляет данные и как именно осуществляется запись. Давайте изменим наш класс, чтобы отвязаться от конкретной реализации и сделать его более универсальным. Для этого добавим зависимость от интерфейса кэширования.

```
<?php namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;
```

```
class Repository {

    /**
     * Create a new repository instance.
     *
     * @param  Cache  $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

}
```

Этот код не связан ни с одной внешней библиотекой, в том числе с ядром фреймворка! Контракт не содержит никакой конкретной реализации кэширования, только интерфейс, и вы можете написать любую свою реализацию кэширования - используя внешние библиотеки или нет. Кроме того, теперь вы можете в любой момент легко изменить способ кэширования, просто подав другую реализацию в конструктор класса при регистрации его в сервис-контейнере в методе `register()` вашего сервис-провайдера.

Упрощение кода

Когда все сервисы ядра фреймворка аккуратно определены в простых интерфейсах, очень легко определить, что именно делает тот или иной сервис. **Фактически, контракты являются краткой документацией к API Laravel**.

Кроме того, когда вы у себя в приложении внедряете в классы зависимости от простых интерфейсов, в вашем коде легче разобраться и его проще поддерживать. Вместо того, чтобы заставлять коллег разбираться, какие методы вашего большого и сложного класса можно использовать извне, вы адресуете их к простому и понятному интерфейсу.

Таблица основных контрактов

Вот таблица соответствий контрактов Laravel 5 фасадам Laravel 4 :

Контракт	Фасад Laravel 4.x
Illuminate\Contracts\Auth\Guard	Auth
Illuminate\Contracts\Auth\PasswordBroker	Password
Illuminate\Contracts\Cache\Repository	Cache
Illuminate\Contracts\Cache\Factory	Cache::driver()
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Container\Container	App
Illuminate\Contracts\Cookie\Factory	Cookie
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Encryption\Encrypter	Crypt
Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	
Illuminate\Contracts\Filesystem\Factory	File
Illuminate\Contracts\Filesystem\Filesystem	File
Illuminate\Contracts\Foundation\Application	App

Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Logging\Log	Log
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Queue\Factory	Queue::driver()
Illuminate\Contracts\Queue\Queue	Queue
Illuminate\Contracts\Redis\Database	Redis
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Validation\Factory	Validator::make()
Illuminate\Contracts\Validation\Validator	
Illuminate\Contracts\View\Factory	View::make()
Illuminate\Contracts\View\View	

Использование контрактов

Как получить в своем классе реализацию заданного контракта? Очень просто. Достаточно в аргументах конструктора явно указать в качестве типа аргумента название соответствующего контракта - и при его создании там окажется реализация этого интерфейса. Это происходит за счет того, что почти все классы Laravel (контроллеры, слушатели событий и очередей, роуты, фильтры роутов) регистрируются в сервис-контейнере и в процессе создания экземпляра класса из контейнера в него встраиваются все определенные таким образом зависимости. Фактически все происходит автоматически, вам не приходится задумываться об этом механизме.

Например, взглянем на слушателя событий:

```
<?php namespace App\Events;

use App\User;
use Illuminate\Contracts\Queue\Queue;

class NewUserRegistered {

    /**
     * The queue implementation.
     */
    protected $queue;

    /**
     * Create a new event listener instance.
     *
     * @param  Queue  $queue
     * @return void
     */
    public function __construct(Queue $queue)
    {
```

```
        $this->queue = $queue;
    }

    /**
     * Handle the event.
     *
     * @param  User  $user
     * @return void
     */
    public function fire(User $user)
    {
        // Queue an e-mail to the user...
    }

}
```

Если вы хотите больше узнать о сервис-контейнере, прочтите соответствующий раздел документации.

Введение

Фасады предоставляют «статический» интерфейс (`Foo::bar()`) к классам, доступным в сервис-контейнере. Laravel поставляется со множеством фасадов и вы, вероятно, использовали их, даже не подозревая об этом.

Иногда вам может понадобиться создать собственные фасады для вашего приложения и пакетов (packages), поэтому давайте изучим идею, разработку и использование этих классов.

> **Примечание:** перед погружением в фасады настоятельно рекомендуется как можно детальнее изучить сервис-контейнер Laravel.

Описание

В контексте приложения на Laravel, фасад - это класс, который предоставляет доступ к объекту в контейнере. Весь этот механизм реализован в классе `Facade`. Фасады как Laravel, так и ваши собственные, наследуют этот базовый класс.

Ваш фасад должен определить единственный метод: `getFacadeAccessor`. Его задача - определить, что вы хотите получить из контейнера. Класс `Facade` использует магический метод PHP `__callStatic()` для перенаправления вызовов методов с вашего фасада на полученный объект.

Например, когда вы вызываете `Cache::get()`, Laravel получает объект `CacheManager` из сервис-контейнера и вызывает метод `get` этого класса. Другими словами, фасады Laravel предоставляют удобный синтаксис для использования сервис-контейнера в качестве сервис-локатора (service locator).

Практическое использование

В примере ниже делается обращение к механизму кэширования Laravel. На первый взгляд может показаться, что метод `get` принадлежит классу `Cache`.

```
$value = Cache::get('key');
```

Однако, если вы посмотрите в исходный код класса `Illuminate\Support\Facades\Cache`, то увидите, что он не содержит метода `get`:

```
class Cache extends Facade {

    /**
     * Получить зарегистрированное имя компонента.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }

}
```

Класс Cache наследует класс `Facade` и определяет метод `getFacadeAccessor()`. Как вы помните, его задача - вернуть строковое имя (ключ) привязки объекта в сервис-контейнере.

Когда вы обращаетесь к любому статическому методу фасада `Cache`, Laravel получает объект `cache` из сервис-контейнера и вызывает на нём требуемый метод (в этом случае - `get`).

Таким образом, вызов `Cache::get` может быть записан так:

```
$value = $app->make('cache')->get('key');
```

Импорт фасадов

Помните, если вы используете фасады в контроллерах с указанным пространством имён, вам нужно импортировать классы фасадов. Все фасады находятся в глобальном пространстве имён.

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

use Cache;
```

```
class PhotosController extends Controller {

    /**
     * Get all of the application photos.
     *
     * @return Response
     */
    public function index()
    {
        $photos = Cache::get('photos');

        //
    }

}
```

Создание фасадов

Создать фасад довольно просто. Вам нужны только три вещи:

1. Биндинг (привязка) в сервис-контейнер.
2. Класс-фасад.
3. Настройка для псевдонима фасада.

Посмотрим на следующий пример. Здесь определён класс `PaymentGateway\Payment`.

```
namespace PaymentGateway;

class Payment {

    public function process()
    {
        //
    }

}
```

Нам нужно, чтобы этот класс извлекался из сервис-контейнера, так что давайте добавим для него привязку (binding):

```
App::bind('payment', function()
{
    return new \PaymentGateway\Payment;
});
```

Самое лучшее место для регистрации этой связки - новый сервис-провайдер который мы назовём `PaymentServiceProvider` и в котором мы создадим метод `register`, содержащий код выше. После этого вы можете настроить Laravel для загрузки этого провайдера в файле `config/app.php`.

Дальше мы можем написать класс нашего фасада:

```
use Illuminate\Support\Facades\Facade;

class Payment extends Facade {

    protected static function getFacadeAccessor() { return 'payment'; }

}
```

Наконец, по желанию можно добавить псевдоним (alias) для этого фасада в массив `aliases` файла настроек `config/app.php`- тогда мы сможем вызывать метод `process` на классе `Payment`.

```
Payment::process();
```

Об автозагрузке псевдонимов

В некоторых случаях классы в массиве `aliases` не доступны из-за того, что PHP не загружает неизвестные классы в подсказках типов. Если `\ServiceWrapper\ApiTimeoutException` имеет псевдоним `ApiTimeoutException`, то блок `catch(ApiTimeoutException $e)`, помещённый в любое пространство имён, кроме `\ServiceWrapper`, никогда не «поймает» это исключение, даже если оно было возбуждено внутри него. Аналогичная проблема возникает в классах, которые содержат подсказки типов на неизвестные (неопределённые) классы. Единственное решение - не использовать псевдонимы и вместо них в начале каждого файла писать `use` для ваших классов.

Фасады для тестирования

Юнит-тесты играют важную роль в том, почему фасады делают именно то, то они делают. На самом деле возможность тестирования - основная причина, по которой фасады вообще существуют. Эта тема подробнее раскрыта в соответствующем разделе документации - фасады-заглушки.

Facade Class Reference

Ниже представлена таблица соответствий фасадов Laravel и классов, лежащих в их основе.

Фасад	Класс	Имя в IoC
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Console\Application	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	`cache`
Config	Illuminate\Config\Repository	`config`
Cookie	Illuminate\Cookie\CookieJar	`cookie`
Crypt	Illuminate\Encryption\Encrypter	`encrypter`
DB	Illuminate\Database\DatabaseManager	`db`
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Form	Illuminate\Html\FormBuilder	`form`
Hash	Illuminate\Hashing\HasherInterface	`hash`
HTML	Illuminate\Html\HtmlBuilder	`html`
Input	Illuminate\Http\Request	`request`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Paginator	Illuminate\Pagination\Factory	`paginator`
Paginator (Instance)	Illuminate\Pagination\Paginator	
Password	Illuminate\Auth\Passwords\PasswordBroker	`auth.reminder`

Queue	Illuminate\Queue\QueueManager	`queue`
Queue (Instance)	Illuminate\Queue\QueueInterface	
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\Database	`redis`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Support\Facades\Response	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	
SSH	Illuminate\Remote\RemoteManager	`remote`
SSH (Instance)	Illuminate\Remote\Connection	
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`
View (Instance)	Illuminate\View\View	

Введение

Когда вы используете какую-то вещь, вы получаете гораздо больше удовольствия от неё, когда понимаете, как она работает. Разработка приложений не исключение. Когда вы понимаете, как именно функционирует ваше средство разработки, вы можете использовать его более уверенно - не просто копируя "магические" куски кода из мануала или других приложений, а точно зная, что вы хотите получить.

Цель этого документа - дать вам хороший высокоуровневый взгляд на то, как работает фреймворк Laravel.

Не расстраивайтесь, если поначалу вам будут непонятны какие-то термины. Просто попробуйте получить базовое понимание происходящего, а ваши знания будут расти по мере того как вы будете изучать другие части документации Laravel.

Обзор

Первые шаги

Точка входа в приложение Laravel - файл `public/index.php`. Все запросы веб-сервер (Apache или Nginx) направляет сюда. Файл не содержит много кода, это просто точка, откуда начинает загружаться фреймворк и где отдается контент браузеру.

`index.php` загружает созданный Composer-ом автозагрузчик классов и при помощи `bootstrap/app.php` создает `$app` - объект приложения, или сервис-контейнер.

Ядро обработки HTTP- и консольных запросов

Далее запрос поступает или в ядро обработки HTTP-запросов или в ядро обработки консольных запросов - в зависимости от того, откуда пришел запрос. Для примера остановимся на HTTP-ядре, `app/Http/Kernel.php`.

HTTP-ядро наследуется от класса `Illuminate\Foundation\Http\Kernel`, в котором определён массив `bootstrappers` с классами, которые должны запускаться перед обработкой запроса. Там есть обработчики ошибок, класс, конфигурирующий логирование, классы, реализующие загрузку конфигов, получение названия среды выполнения и выполнения других задач, которые должны быть исполнены перед обработкой запроса.

В HTTP-ядре также определён список middleware (посредников), через которые должен пройти запрос и быть разрешённым к исполнению. Посредники реализуют чтение и запись HTTP-сессии, определяют, находится ли приложение в режиме обслуживания, проверяют CSRF-токен и т.п.

HTTP-ядро - это как некий черный ящик. Принимает на вход `Request` (запрос) и отдает `Response` (ответ).

Сервис-провайдеры

Один из самых важных моментов в первой фазе работы фреймворка - загрузка сервис-провайдеров вашего приложения. Список загружаемых сервис-провайдеров находится в файле `config/app.php` в массиве `providers`. В процессе загрузки выполняется метод `register` каждого сервис-провайдера, а когда все они будут загружены - в метод `boot`, также у каждого сервис-провайдера.

Выполнение запроса

Как только все сервис провайдеры зарегистрированы и приложение загружено, `Request` поступает в роутер для обработки. Там фреймворк принимает решение, в какой именно роут попадает запрос, через какую цепочку посредников он должен пройти и в какой метод какого контроллера попасть.

Фокус на сервис-провайдеры !

Сервис-провайдер - ключевая вещь фреймворка, ключ к пониманию процесса загрузки (bootstrapping) вашего приложения.

Ваши сервис-провайдеры находятся в папке `app/Providers`. Дефолтный стартовый сервис-провайдер приложения `AppServiceProvider` не содержит почти ничего, его должны заполнить вы - регистрацией сервис-провайдеров модулей, биндингами в сервис-контейнер своих классов и фасадов и т.д.

Introduction

Дефолтная структура приложения Laravel спроектирована таким образом, чтобы стать удобной отправной точкой и для маленьких, и для больших приложений. И, разумеется, вы можете изменить эту структуру и организовать приложение так, как вам нравится - Laravel не накладывает почти никаких ограничений на то, где именно должен находиться тот или иной класс - лишь бы Composer смог его загрузить.

Корневой каталог

В корне свежеустановленного фреймворка вы можете видеть следующие каталоги:

app - здесь, как вы догадываетесь, располагается собственно ваше приложение. Ниже мы рассмотрим содержимое этого каталога подробнее.

bootstrap - содержит файлы, которые осуществляют первоначальную загрузку (bootstraping) фреймворка и настраивают автозагрузку классов.

config - здесь находятся конфигурационные файлы приложения.

database - каталог для файлов миграций БД и "посева" данных.

public - является DocumentRoot домена вашего приложения и содержит статические файлы - css, js, изображения и т.п.

resources - здесь находятся шаблоны (Views), файлы локализации и, если таковые имеются, рабочие файлы LESS, SASS и js-приложения на фреймворках типа AngularJS или Ember, которые потом собираются внешним инструментом в папку public.

storage - этот каталог должен иметь права для записи в него извне и в нём Laravel хранит скомпилированные Blade-шаблоны, файлы сессии, файловый кэш и другие сгенерированные файлы, нужные для работы.

test - каталог для юнит-тестов

vendor - в этот каталог Composer устанавливает пакеты, указанные в composer.json.

Папка приложения

В каталоге app находятся классы вашего Laravel-приложения. По умолчанию, этот каталог имеет неймспейс App и классы в нём автозагружаются согласно стандарту PRS-4. Вы можете сменить это имя при помощи Artisan-команды app:name.

Внутри находятся несколько дополнительных каталогов, таких как Console, Http и Providers. Первые два каталога, как следует из названия, содержат классы, предоставляющие API к вашему приложению по протоколам CLI (командная строка) и HTTP (работа через браузер). В Console находятся классы Artisan-команд, а в Http - контроллеры, фильтры и реквесты, т.е. классы валидации пользовательского ввода. Такой подход должен подтолкнуть новичков к отходу от общепринятого, но вредного подхода писать весь код в контроллерах, и абстрагировать логику приложения от метода обращения к нему.

Каталог Commands содержит классы команд вашего приложения. Команды представляют собой задания, которые можно ставить в очередь, а так же задачи, которые можно запускать синхронно с процессом обработки запроса.

Каталог Events содержит классы событий. Конечно, использование классов для представления событий не обязательно, но если вы решите делать именно так, то они будут помещаться в этот каталог при создании с помощью Artisan CLI.

Каталог Handlers содержит классы обработчиков для команд и событий. Обработчики получают команду или событие и выполняют необходимые действия в ответ на инициирование этой команды или события.

Каталог Services содержит различные вспомогательные механизмы вашего приложения (хелперы и сервисы). Например, сервис Registrar, включённый в Laravel, отвечает за валидацию данных и регистрацию пользователей вашего приложения. Другим примером может быть сервис для взаимодействия с какими-либо внешними API, системами статистики или даже другими сервисами, которые получают данные от вашего приложения.

Каталог Exceptions содержит обработчики исключений, а так же здесь следует размещать сами классы исключений, которые используются в вашем приложении.

> **Примечание:** В каталоге app можно генерировать соответствующие классы командами

`make:controller`, `make:filter`, `make:request`, `make:console` и `make:provider`.

Изменения названия неймспейса

Как сказано выше, по умолчанию неймспейс, в котором располагается приложение Laravel называется `App`. Вы можете сменить его соответствующей командой:

```
php artisan app:name SocialNet
```

Во всех файлах классов в каталоге `app` название корневого неймспейса будет изменено на `SocialNet`

Введение

Laravel позволяет сделать аутентификацию очень простой. Фактически, почти всё уже готово к использованию «из коробки». Настройки аутентификации находятся в файле `config/auth.php`, который содержит несколько хорошо документированных опций для настройки механизма аутентификации.

По умолчанию, Laravel использует модель `App\User` в каталоге app. Эта модель может использоваться вместе с драйвером аутентификации на базе Eloquent.

В Laravel уже включены все необходимые миграции для создания аутентификации, но при самостоятельном создании схемы БД не забудьте про два обязательных поля в таблице `user` (или аналогичной):

- поле с паролем длиной минимум в 60 символов,
- поле `remember_token` для хранения идентификаторов «запомнить меня» длиной в 100 символов (можно создать методом `$table->rememberToken();` в миграции).

Если ваше приложение не использует Eloquent, вы можете использовать драйвер аутентификации `database`, который работает через конструктор запросов.

Аутентификация пользователей

В Laravel уже есть два контроллера, относящиеся к механизму аутентификации. Контроллер `AuthController` обрабатывает регистрацию пользователей и вход в приложение, тогда как контроллер `PasswordController` содержит механизмы для сброса забытых паролей у существующих пользователей.

Каждый из этих контроллеров использует трейты для подключения необходимых методов. Как правило, вам не нужно менять код этих контроллеров. Шаблоны, используемые этими контроллерами, находятся в каталоге `resources/views/auth` и вы можете свободно изменять их так, как вам нужно.

Регистрация пользователей

Для редактирования набора полей формы, которую заполняет пользователь при регистрации, необходимо изменить класс `App\Services\Registrar`, который так же отвечает за валидацию введённых данных и создание новых пользователей.

Метод `validator` класса `Registrar` содержит правила валидации для новых пользователей, тогда как метод `create` отвечает непосредственно за создание новой записи о пользователе в БД вашего приложения. Вы можете изменять код этих методов при необходимости. `Registrar` вызывается в контроллере `AuthController` через метод, находящийся в трейте `AuthenticatesAndRegistersUsers`.

Ручная аутентификация

Если вы не хотите использовать механизм, предоставляемый контроллером `AuthController`, то вы должны использовать сервис аутентификации напрямую. Не волнуйтесь, это не сложно! Для начала, давайте посмотрим на метод `attempt`:

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

use Auth;
use Illuminate\Routing\Controller;

class AuthController extends Controller {

    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password]))
        {
            return redirect()->intended('dashboard');
        }
```

```
    }

}
```

Метод `attempt` принимает массив «ключ-значение» в качестве первого аргумента. Значение ключа `password` будет захэшировано. Другие значения массива используются для поиска пользователя в таблице БД. В примере выше, пользователь будет выбираться по полю `email`. Если пользователь будет найден, то хэшированный пароль из БД будет сравнён с хэшированным значение поля `password` из переданного массива. Если два этих хэша совпадут, то для пользователя будет создана новая аутентифицированная сессия.

Метод `attempt` возвращает `true`, если аутентификация прошла успешно, и `false` в противном случае.

> **Примечание:** В примере выше, поле `email` не является обязательным, оно выбрано для примера. Вы должны использовать то название колонки, в котором хранится логин в приложении. Как правило, это «username».

Метод `intended` возвращает пользователя на тот адрес, доступ к которому он хотел получить, прежде чем его поймал фильтр аутентификации. В качестве параметра, в этот метод можно передать резервный адрес, если запрашиваемый адрес недоступен.

Аутентификация пользователя с дополнительными условиями

Вы можете добавить дополнительные условия в аутентификационный запрос:

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1]))
{
    // Пользователь существует, не забанен и активен.
}
```

Как узнать, что пользователь прошел аутентификацию?

Для проверки, аутентифицирован ли пользователь в вашем приложении, можно использовать метод `check`:

```
if (Auth::check())
{
    // Пользователь аутентифицирован...
}
```

Аутентификация и «запоминание» пользователя

Если вы хотите разрешить пользователям использовать механизм «запомнить меня», вы можете передать булево значение вторым аргументом в метод `attempt`, что позволит сохранить статус аутентификации пользователя либо навсегда, либо до тех пор, пока он сам не выйдет из приложения. Естественно, ваша табличка `users` должна содержать колонку `remember_token`, которая будет использоваться для хранения токена пользователя.

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember))
{
    // Пользователь будет «запомнен»...
}
```

Если вы «запоминаете» пользователя, то можете использовать метод `viaRemember` чтобы определить, был ли пользователь аутентифицирован с использованием этого механизма:

```
if (Auth::viaRemember())
{
    //
}
```

Аутентификация пользователя по ID

Для аутентификации пользователя по его ID существует метод `loginUsingId`:

```
Auth::loginUsingId(1);
```

Проверка прав пользователя без аутентификации

Метод `validate` позволяет проверить права пользователя без фактической аутентификации:

```
if (Auth::validate($credentials))
{
    //
}
```

Аутентификация пользователя на время выполнения текущего запроса

Метод once служит для аутентификация пользователя на время выполнения текущего запроса, при этом не используются сессия и куки:

```
if (Auth::once($credentials))
{
    //
}
```

Ручная аутентификация пользователя

Для принудительной аутентификации пользователя существует метод `login`:

```
Auth::login($user);
```

Это эквивалентно аутентификации с использованием метода `attempt` и передачей параметров пользователя.

Выход из приложения

```
Auth::logout();
```

Конечно, если вы используете встроенные контроллеры Laravel для аутентификации, то в них всё это уже реализовано.

События при аутентификации

При вызове метода `attempt` запускается событие `auth.attempt`. Если аутентификация прошла успешно и пользователь вошёл в приложение, будет запущено событие `auth.login`.

Получение аутентифицированного пользователя

Как только пользователь аутентифицирован, вы можете получить объект пользователя несколькими путями.

Во-первых, с помощью фасада `Auth`:

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

use Illuminate\Routing\Controller;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile()
    {
        if (Auth::user())
        {
            // Auth::user() возвращает объект пользователя...
        }
    }

}
```

Во-вторых, используя метод `user` класса `Illuminate\Http\Request`:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile(Request $request)
    {
```

```
        if ($request->user())
        {
            // $request->user() возвращает объект пользователя...
        }
    }

}
```

В третьих, можно использовать мощь сервис-контейнера, указав в качестве аргумента в конструкторе или методе контракт `Illuminate\Contracts\Auth\Authenticatable`:

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

use Illuminate\Routing\Controller;
use Illuminate\Contracts\Auth\Authenticatable;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile(Authenticatable $user)
    {
        // $user - это объект пользователя...
    }

}
```

Ограничение доступа к роутам

Вы можете использовать посредников (middleware) для ограничения доступа к роутам. В Laravel уже есть посредник `auth`, который находится в файле `app\Http\Middleware\Authenticate.php`. Всё, что вам нужно - указать его в описании нужного роута:

```
// Если роут описан как замыкание...

Route::get('profile', ['middleware' => 'auth', function()
{
    // Доступ разрешён только аутентифицированным пользователям...
}]);

// Или как контроллер...

Route::get('profile', ['middleware' => 'auth', 'uses' => 'ProfileController@show']);
```

Аутентификация на основе HTTP Basic

Аутентификация на основе HTTP Basic позволяет аутентифицировать пользователей быстро и без отдельной страницы входа. Для этого надо указать посредника `auth.basic` в описании нужного роута:

Защита роута при помощи HTTP Basic

```
Route::get('profile', ['middleware' => 'auth.basic', function()
{
    // Доступ разрешён только аутентифицированным пользователям...
}]);
```

По умолчанию, посредник `basic` использует поле `email` из таблицы пользователей в качестве идентификатора пользователя.

Настройка Stateless HTTP Basic фильтра

Так же можно использовать аутентификацию на основе HTTP Basic без создания сессии и идентификационной куки, что часто используется для аутентификации через API. Для этого нужно создать посредника, который будет вызывать метод `onceBasic`:

```
public function handle($request, Closure $next)
{
    return Auth::onceBasic() ?: $next($request);
}
```

Если вы используете PHP в режиме FastCGI, то аутентификация на основе HTTP Basic может не работать «из коробки». Решается эта проблема добавлением следующих строк в файл `.htaccess`:

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Напоминание и сброс пароля

Модель и таблица

Большинство веб-приложения позволяют пользователям сбросить забытый пароль. Вместо того, чтобы каждый раз изобретать велосипед, Laravel предоставляет удобный механизм для реализации этой возможности.

Для начала, удостоверьтесь, что ваша модель `User` реализует контракт `Illuminate\Contracts\Auth\CanResetPassword` в том случае, если вы не используете стандартную модель `User`, которая уже реализует этот контракт при помощи трейта `Illuminate\Auth\Passwords\CanResetPassword`.

Создание миграции для таблицы с напоминаниями паролей

Далее нужно создать таблицу, хранящую токены-напоминания для сброса пароля. Эта миграция уже включена в Laravel и находится в папке `database/migrations`. Поэтому всё, что вам нужно сделать, это выполнить команду:

```
php artisan migrate
```

Контроллер напоминания паролей

Laravel так же предоставляет контроллер `Auth\PasswordController`, который содержит всю логику для работы со сбросом паролей. Так же есть и начальные шаблоны! Шаблоны находятся в папке `resources/views/auth` и вы можете изменять их так, как вам нужно.

Пользователь получает письмо со ссылкой, обрабатываемой методом `getReset` в контроллере `PasswordController`. Это метод отображает форму, где пользователь указывает новый пароль. После этого пользователь автоматически аутентифицируется и перенаправляется на адрес `/home`. Вы можете задать свой адрес в свойстве `redirectTo` контроллера `PasswordController`:

```
protected $redirectTo = '/dashboard';
```

> **Примечание:** По умолчанию, срок жизни токена ограничен одним часом. Вы можете изменить это, указав нужное время в опции `reminder.expire` в файле `config/auth.php`.

Аутентификация через социальные сети

В добавок к обычной аутентификации, с помощью форм и HTTP Basic, Laravel предоставляет простой и удобный механизм аутентификации через OAuth, используя Laravel Socialite. **Socialite пока что поддерживает аутентификацию только через Facebook, Twitter, Google и GitHub.**

Чтобы начать использовать Socialite, добавьте этот пакет в ваш файл `composer.json`:

```
"laravel/socialite": "~2.0"
```

После, зарегистрируйте провайдер `Laravel\Socialite\SocialiteServiceProvider` в файле `config/app.php`. Вы так же можете зарегистрировать фасад `Socialize`:

```
'Socialize' => 'Laravel\Socialite\Facades\Socialite',
```

Далее вам необходимо указать параметры для того сервиса OAuth, который вы используете. Это можно сделать в файле `config/services.php`. Пока доступно четыре сервиса: `facebook`, `twitter`, `google` и `github`. Пример:

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

После этого нужно добавить два роута: один для перенаправления пользователя к провайдеру OAuth, второй для получения ответа от провайдера после аутентификации пользователя. Пример с использованием фасада `Socialize`:

```
public function redirectToProvider()
{
    return Socialize::with('github')->redirect();
}
```

```
public function handleProviderCallback()
{
    $user = Socialize::with('github')->user();

    // $user->token;
}
```

Метод `redirect` выполнит перенаправление к провайдеру OAuth, а метод `user` получит информацию о пользователе из ответа провайдера OAuth. Перед перенаправлением, вы можете указать области доступа:

```
return Socialize::with('github')->scopes(['scope1', 'scope2'])->redirect();
```

Как только вы получите объект пользователя, можно получить пользовательские данные:

Получение данных пользователя

```
$user = Socialize::with('github')->user();

// Провайдеры, использующие OAuth v.2.0
$token = $user->token;

// Провайдеры, использующие OAuth v.1.0
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// Все провайдеры
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

git eb6dd6023431edab59f665b768a6b4618e03e74f

Настройка

Laravel предоставляет унифицированное API для различных систем кэширования. Настройки кэша содержатся в файле `config/cache.php`. Там вы можете указать драйвер, который будет использоваться для кэширования. Laravel "из коробки" поддерживает многие популярные системы, такие как Memcached и Redis.

Этот файл также содержит множество других настроек, которые в нём же документированы, поэтому обязательно ознакомьтесь с ними. По умолчанию Laravel настроен для использования драйвера `file`, который хранит сериализованные объекты кэша в файловой системе. Для больших приложений рекомендуется использование систем кэширования в памяти - таких как `Memcached` или APC. Вы так же можете создать несколько разных конфигурций для одного драйвера.

Прежде чем использовать Redis, необходимо установить пакет `predis/predis` версии `~1.0` через Composer.

Использование кэша

Запись нового элемента в кэш

```
Cache::put('key', 'value', $minutes);
```

Использование объектов Carbon для установки времени жизни кэша

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

Запись элемента, если он не существует

```
Cache::add('key', 'value', $minutes);
```

Метод add возвращает `true`, если производится запись элемента в кэш. Иначе, если элемент уже есть в кэше, возвращается `false`.

Проверка существования элемента в кэше

```
if (Cache::has('key'))
{
    //
}
```

Чтение элемента из кэша

```
$value = Cache::get('key');
```

Чтение элемента или возвращение значения по умолчанию

```
$value = Cache::get('key', 'default');

$value = Cache::get('key', function() { return 'default'; });
```

Запись элемента на постоянное хранение

```
Cache::forever('key', 'value');
```

Иногда вам может понадобиться получить элемент из кэша или сохранить его там, если он не существует. Вы можете сделать это методом `Cache::remember`:

```
$value = Cache::remember('users', $minutes, function()
{
    return DB::table('users')->get();
});
```

Вы также можете совместить `remember` и `forever`:

```
$value = Cache::rememberForever('users', function()
{
```

```
    return DB::table('users')->get();
});
```

Обратите внимание, что все кэшируемые данные сериализуются, поэтому вы можете хранить любые типы данных.

Изъятие элемента из кэша

Если понадобится получить элемент из кэша, а потом удалить его, можно воспользоваться методом `pull`:

```
$value = Cache::pull('key');
```

Удаление элемента из кэша

```
Cache::forget('key');
```

Доступ к определённому хранилищу

Если вы используете несколько хранилищ для кэша (с одинаковыми или разными драйверами) - вы можете обратиться к конкретному хранилищу следующим образом:

```
$value = Cache::store('foo')->get('key');
```

Увеличение и уменьшение значений

Все драйверы, кроме `file` и `database`, поддерживают операции инкремента и декремента.

Увеличение числового значения:

```
Cache::increment('key');

Cache::increment('key', $amount);
```

Уменьшение числового значения:

```
Cache::decrement('key');

Cache::decrement('key', $amount);
```

Тэги кэша

> **Примечание:** тэги кэша не поддерживаются драйверами `file` и `database`. Кроме того, если вы используете мультитэги для "вечных" элементов кэша (сохраненных как forever), наиболее подходящим с точки зрения производительности будет драйвер типа `memcached`, который автоматически очищает устаревшие записи.

Работа с тэгами кэша

При помощи тэгов вы можете объединять элементы кэша в группы, а затем очищать всю группу целиком по названию тэга. Для работы с кэшем с тэгами используйте метод `tags`.

Элементу кэша можно присвоить один или несколько тэгов кэша. Список тэгов можно указать либо перечислив через запятую, либо массивом:

```
Cache::tags('people', 'authors')->put('John', $john, $minutes);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

Любой метод для записи в кэш можно использовать в связке с тэгами, включая `remember`, `forever` и `rememberForever`. Элемент кэша также можно получить из кэша с тэгом, так же как и использовать другие методы кэша, такие как `increment` и `decrement`.

Получение элементов из кэша с тэгами

Чтобы получить элемент кэша, вы должны указать все тэги, под которыми он был сохранен:

```
$anne = Cache::tags('people', 'artists')->get('Anne');

$john = Cache::tags(['people', 'authors'])->get('John');
```

Вы можете очистить все элементы по тэгу или списку тэгов. Например, это выражение удалит все элементы кэша с тэгом или `people`, или `authors`, или в обоих сразу. Таким образом, и "Anne", и "John" будут удалены из кэша:

```
Cache::tags('people', 'authors')->flush();
```

Для сравнения, это выражение удалит только элементы с тэгом `authors`, таким образом "John" будет удален, а "Anne" нет:

```
Cache::tags('authors')->flush();
```

Кэширование в базе данных

Перед использовании драйвера `database` вам понадобится создать таблицу для хранения элементов кэша. Ниже приведён пример её структуры в виде миграции Laravel:

```
Schema::create('cache', function($table)
{
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

Введение

Класс `Illuminate\Support\Collection` - гибкая и удобная обёртка для работы с массивами.

Для примера, взгляните на код ниже. При помощи хэлпера `collect` мы создаём коллекцию из массива:

```
$collection = collect(['taylor', 'abigail', null])->map(function($name)
{
    return strtoupper($name);
})
->reject(function($name)
{
    return is_null($value);
});
```

Как вы можете видеть, класс `Collection` позволяет строить цепочки вызовов для операций типа `map` и `reduce` над заданным массивом данных. В основном каждый метод класса `Collection` возвращает новый объект класса `Collection`.

Основы использования

Создание коллекции

Чтобы создать коллекцию из массива, воспользуйтесь функцией-хэлпером или методом `make`:

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

$collection = Collection::make([1, 2, 3]);
```

Там, где методы Eloquent возвращают не один, а несколько объектов - возвращается коллекция. Однако, коллекции предоставляют слишком мощный функционал, чтобы оставить его только для Eloquent-моделей, попробуйте применить класс `Collection` у себя в приложении.

Посмотрите, что могут коллекции

Вместо того, чтобы приводить здесь огромный скучный список методов коллекции, мы отсылаем вас к более удобной документации API этого класса.

Введение

Командная шина в Laravel - это удобный способ инкапсуляции (изолирования) задач вашего приложения в простые и понятные «команды». Чтобы разобраться в назначении команд, давайте представим, что мы пишем приложение, в котором мы будем продавать подкасты нашим пользователям.

Когда пользователь приобретает подкаст, нам нужно произвести несколько действий: мы должны снять деньги с его карты, добавить запись в БД, что покупка успешно состоялась и отослать email с подтверждением покупки. Возможно, еще нам нужно провести несколько проверок - а разрешено ли этому пользователю вообще покупать этот конкретный подкаст.

Обычно мы размещаем всю эту логику в контроллере, однако такой подход имеет несколько недостатков. Во-первых, класс контроллера обычно содержит методы для обработки нескольких HTTP-запросов, и располагая логику в каждом методе мы делаем контроллеры излишне большими, сложно читаемыми и сложно поддерживаемыми. Во-вторых, такой код трудно использовать вне контекста контроллера и HTTP-запроса (например, если мы хотим выполнять покупку подкастов из командной строки или очереди). В-третьих, это затрудняет тестирование, так как мы вынуждены эмулировать HTTP-запрос и разбирать HTTP-ответ.

Поэтому вместо помещения этой логики в контроллер мы оформим её в так называемую команду и назовём её `PurchasePodcast`.

Создание команды

Создаем команду при помощи artisan-команды `make:command`:

```
php artisan make:command PurchasePodcast
```

Созданный этой командой класс помещается в папку `app/Commands`. По умолчанию команда содержит два метода - конструктор и метод `handle`. При помощи конструктора вы можете добавить нужные зависимости в класс команды, а метод `handle` собственно исполняет команду. Например:

```
class PurchasePodcast extends Command implements SelfHandling {

    protected $user, $podcast;

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct(User $user, Podcast $podcast)
    {
        $this->user = $user;
        $this->podcast = $podcast;
    }

    /**
     * Execute the command.
     *
     * @return void
     */
    public function handle()
    {
        // Пишем функцонал покупки подкаста здесь...

        event(new PodcastWasPurchased($this->user, $this->podcast));
    }

}
```

Метод `handle` тоже может принимать зависимости в аргументах (type hinting), как и конструктор. Как и конструктору, они будут поданы на вход автоматически при помощи IoC контейнера.

```
    /**
     * Execute the command.
     *
     * @return void
```

```
     */
    public function handle(BillingGateway $billing)
    {
        // Пишем функцонал покупки подкаста здесь...
    }
```

Выполнение команд

Мы создали команду, как теперь запустить её? Конечно, мы можем выполнить метод `handle` нашего класса, однако лучше запускать его через командную шину Laravel. О преимуществах такого подхода будет рассказано ниже.

Если вы взглянете на базовый контроллер вашего приложения, который расширяют ваши собственные контроллеры, вы увидите там трейт `DispatchesCommands`. Этот трейт позволяет запускать команды при помощи метода `dispatch`. Например:

```
public function purchasePodcast($podcastId)
{
    $this->dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
}
```

Командная шина исполняет команду и берёт на себя всю рутину по обеспечению класса команды всеми необходимыми зависимостями, перечисленными в аргументах конструктора класса команды и аргументах метода `handle`.

Вы можете задействовать командную шину в любом вашем классе - для этого добавьте трейт `Illuminate\Foundation\Bus\DispatchesCommands`. Если вы хотите принимать инстанс командной шины в конструкторе, то укажите в аргументах (type hint) объект `Illuminate\Contracts\Bus\Dispatcher`. И наконец вы можете просто использовать фасад Bus для запуска команды:

```
    Bus::dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
```

Передача аргументов из запроса

Практически всегда вам понадобится передать данные HTTP-запроса (например, выборочное из $_POST) в команду. Вместо того, чтобы заставлять вас вручную описывать каждый запрос, Laravel предлагает нечто более автоматизированное. Посмотрите на метод `dispatchFrom` трейта `DispatchesCommands`:

```
$this->dispatchFrom('Command\Class\Name', $request);
```

Этот метод смотрит конструктор класса, имя которого передано первым аргументом, и вынимает из переменной $request (это HTTP-запрос или просто любой объект типа `ArrayObject`) те ключи, которые совпадают с названием переменных в аргументе конструктора. Например, если в аргументах конструктора есть переменная `$firstName`, то ей присвоится значение `firstName` HTTP-запроса.

Вы можете передать третьим аргументом массив значений по умолчанию:

```
$this->dispatchFrom('Command\Class\Name', $request, [
    'firstName' => 'Taylor',
]);
```

Запуск команды в фоне

Командная шина может применяться не только для немедленного запуска задач в текущем запросе, но и помещать команды в очередь для того, чтобы запустить их в отдельном процессе. Таким образом, командная шина может быть основным инструментом для работы с очередями.

Для создания команды, которая будет запускаться в фоне, добавьте флаг `--queued`:

```
php artisan make:command PurchasePodcast --queued
```

Созданный класс будет наследовать интерфейс `Illuminate\Contracts\Queue\ShouldBeQueued` и иметь трейт `SerializesModels`. Этот функционал позволит команде добавляться в очередь для последующего запуска слушателем очереди, а также добавит возможность сериализовать и десериализовать Eloquent-модели.

Если у вас уже есть созданная команда и вы хотите сделать её работающей в фоне, просто вручную добавьте `implements Illuminate\Contracts\Queue\ShouldBeQueued`. Этот интерфейс не содержит обязательных методов и является просто индикатором для командной шины. После этого метод `dispatch` вместо того, чтобы запустить команду, поместит её в очередь для последующего запуска в фоне.

Чтобы узнать поподробнее о том, как в Laravel осуществляется запуск задач в фоне, обратитесь к документации по очередям.

Конвейеры команд

До того, как команда будет передена диспетчеру, вы можете предать её по конвейеру другим классам. Конвейеры команд работают также, как и HTTP-посредники. Например, можно завернуть все операции, выполняемые командой, в транзакцию или просто записать в лог факт её выполнения.

Для добавления конвейера к вашей командной шине, вызовите метод `pipeThrough` диспетчера в своём методе `App\Providers\BusServiceProvider::boot`:

```
$dispatcher->pipeThrough(['UseDatabaseTransactions', 'LogCommand']);
```

Конвейер команд описывается в методе `handle`:

```
class UseDatabaseTransactions {

    public function handle($command, $next)
    {
        return DB::transaction(function() use ($command, $next)
        {
            return $next($command);
        }
    }

}
```

Классы, которые необходимо задействовать в конвейере, ищутся и загружаются через сервис-контейнер, поэтому вы можете указывать любые зависимости в их конструкторах.

В качестве элемента конвейера так же можно использовать замыкания:

```
$dispatcher->pipeThrough([function($command, $next)
{
    return DB::transaction(function() use ($command, $next)
    {
        return $next($command);
    }
}]);
```

Managers и Factory

Laravel содержит несколько классов `Manager`, которые управляют созданием компонентов, основанных на драйверах. Эти компоненты включают в себя кэш, сессии, авторизацию и очереди. Класс-управляющий ответственнен за создание конкретной реализации драйвера в зависимости от настроек приложения. Например, класс `CacheManager` может создавать объекты-реализации APC, Memcached, Native и различных других драйверов.

Каждый из этих управляющих классов имеет метод `extend`, который может использоваться для простого добавления новой реализации драйвера. Мы поговорим об этих управляющих классах ниже, с примерами о том, как добавить собственный драйвер в каждый из них.

> **Примечание:** посветите несколько минут изучению различных классов `Manager`, которые поставляются с Laravel, таких как `CacheManager` и `SessionManager`. Знакомство с их кодом поможет вам лучше понять внутреннюю работу Laravel. Все классы-управляющие наследуют базовый класс `Illuminate\Support\Manager`, который реализует общую полезную функциональность для каждого из них.

Cache

Для расширения подсистемы кэширования мы используем метод `extend` класса `CacheManager`, который используется для привязки стороннего драйвера к управляющему классу и является общим для всех таких классов. Например, для регистрации нового драйвера кэша с именем "mongo" нужно будет сделать следующее:

```
Cache::extend('mongo', function($app)
{
    return Cache::repository(new MongoStore);
});
```

Первый параметр, передаваемый методу `extend` - имя драйвера. Это имя соответствует значению параметра `driver` файла настроек `config/cache.php`. Второй параметр - функция-замыкание, которая должна вернуть объект типа `Illuminate\Cache\Repository`. Замыкание получит параметр `$app` - объект `Illuminate\Foundation\Application`, IoC-контейнер.

Этот вызов `Cache::extend` можно делать в методе `boot()` сервис-провайдера `App\Providers\AppServiceProvider`, или другого вашего сервис-провайдера.

Для создания стороннего драйвера для кэша мы начнём с реализации интерфейса `Illuminate\Contracts\Cache\Store`. Итак, наша реализация MongoDB будет выглядеть примерно так:

```
class MongoStore implements Illuminate\Contracts\Cache\Store {

    public function get($key) {}
    public function put($key, $value, $minutes) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}

}
```

Нам только нужно реализовать каждый из этих методов с использованием подключения к MongoDB. Как только мы это сделали, можно закончить регистрацию нового драйвера:

```
Cache::extend('mongo', function($app)
{
    return Cache::repository(new MongoStore);
});
```

Если вы задумались о том, куда поместить ваш новый дравер - подумайте о том, чтобы сделать отдельный модуль и распространять его через Packagist. Либо вы можете создать пространство имён `Extensions` вместе с соответствующей папкой в папке app. Впрочем, так как Laravel не имеет жёсткой структуры папок, вы можете организовать свои файлы, как вам удобно.

Сессии

Расширение системы сессий Laravel собственным драйвером так же просто, как и расширение драйвером кэша. Мы вновь используем метод extend для регистрации собственного кода:

```
Session::extend('mongo', function($app)
{
    // Вернуть объект, реализующий SessionHandlerInterface
});
```

Где расширять Session

Наиболее подходящее место - метод boot сервис-провайдера AppServiceProvider.

Написание расширения

Заметьте, что наш драйвер сессии должен реализовывать интерфейс SessionHandlerInterface. Он включен в ядро PHP 5.4+. Если вы используете PHP 5.3, то Laravel создаст его для вас, что позволит поддерживать совместимость будущих версий. Этот интерфейс содержит несколько простых методов, которые нам нужно написать. Заглушка драйвера MongoDB выглядит так:

```
class MongoHandler implements SessionHandlerInterface {

    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}

}
```

Эти методы не так легки в понимании, как методы драйвера кэша (StoreInterface), поэтому давайте пробежимся по каждому из них подробнее:

- Метод open обычно используется при открытии системы сессий, основанной на файлах. Laravel поставляется с драйвером native, который использует стандартное файловое хранилище PHP, вам почти никогда не понадобится добавлять что-либо в этот метод. Вы можете всегда оставить его пустым. Фактически, это просто тяжелое наследие плохого дизайна PHP, из-за которого мы должны написать этот метод (мы обсудим это ниже).
- Метод close, аналогично методу open, обычно также игнорируется. Для большей части драйверов он не требуется.
- Метод read должен вернуть строку - данные сессии, связанные с переданным $sessionId. Нет необходимости сериализовать объекты или делать какие-то другие преобразования при чтении или записи данных сессии в вашем драйвере - Laravel делает это автоматически.
- Метод write должен связать строку $data с данными сессии с переданным идентификатором $sessionId, сохранив её в каком-либо постоянном хранилище, таком как MongoDB, Dynamo и др.
- Метод destroy должен удалить все данные, связанные с переданным $sessionId, из постоянного хранилища.
- Метод gc должен удалить все данные, которые старее переданного $lifetime (unixtime секунд). Для самоочищающихся систем вроде Memcached и Redis этот метод может быть пустым.

Как только интерфейс SessionHandlerInterface реализован, мы готовы к тому, чтобы зарегистрировать новый драйвер в управляющем классе Session:

```
Session::extend('mongo', function($app)
{
    return new MongoHandler;
});
```

Как только драйвер сессии зарегистрирован мы можем использовать его имя mongo в нашем файле настроек config/session.php.

> **Подсказка:** если вы написали новый драйвер сессии, поделитесь им на Packagist!

Аутентификация

Механизм аутентификации может быть расширен тем же способом, что и кэш и сессии. Мы используем метод extend, с которым вы уже знакомы:

```
Auth::extend('riak', function($app)
{
    // Вернуть объект, реализующий Illuminate\Contracts\Auth\UserProvider
});
```

Реализация `UserProvider` ответственна только за то, чтобы получать нужный объект, реализующий `Illuminate\Contracts\Auth\Authenticatable` из постоянного хранилища, такого как MySQL, Riak и др. Эти два интерфейса позволяют работать механизму авторизации Laravel вне зависимости от того, как хранятся пользовательские данные и какой класс используется для их представления.

Давайте посмотрим на контракт `UserProvider`:

```
interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);

}
```

Метод `retrieveById` обычно получает числовой ключ, идентифицирующий пользователя - такой, как первичный ключ в MySQL. Метод должен возвращать объект, реализующий `Authenticatable`, соответствующий переданному ID.

Метод `retrieveByToken` запрашивает пользователя по уникальному `$identifier` и "запомнить меня"-токену, который хранится в столбце `remember_token`. Метод должен возвращать объект, реализующий `Authenticatable`

The `updateRememberToken` method updates the `$user` field `remember_token` with the new `$token`. The new token can be either a fresh token, assigned on successfull "remember me" login attempt, or a null when user is logged out.

Метод `updateRememberToken` обновляет у `$user` поле `remember_token` новым `$token`.

Метод `retrieveByCredentials` получает массив данных, ктоорые были переданы методу `Auth::attempt` при попытке входа в систему. Этот метод должен запросить своё постоянное хранилище на наличие пользователя с совпадающими данными. Обычно этот метод выполнит SQL-запрос с проверкой на `$credentails['username']`. **Этот метод не должен производить сравнение паролей или выполнять вход.**

Метод `validateCredentials` должен сравнить переданный объект пользователя `$user` с данными для входа `$credentials` для того, чтобы его авторизовать. К примеру, этот метод может сравнивать строку `$user->getAuthPassword()` с результатом вызова `Hash::make` а строке `$credentials['password']`.

Теперь, когда мы узнали о каждом методе интерфейса `UserProviderInterface`давайте посмотрим на интерфейс `Authenticatable`. Как вы помните, поставщик должен вернуть реализацию этого интерфейса из своих методов `retrieveById` и `retrieveByCredentials`.

```
interface Authenticatable {

    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();

}
```

Это простой интерфейс. Метод `getAuthIdentifier` должен просто вернуть "первичный ключ" пользователя. Если используется хранилище MySQL, то это будет автоматическое числовое поле - первичный ключ. Метод `getAuthPassword` должен вернуть хэшированный пароль. Этот интерфейс позволяет системе авторизации работать с любым классом пользователя, вне зависимости от используемой ORM или хранилища данных. Изначально Laravel содержит класс `User` в папке `app` который реализует этот интерфейс, поэтому мы можете обратиться к этому классу, чтобы увидеть пример реализации.

Наконец, как только мы написали класс-реализацию `UserProvider`, у нас готово для регистрации расширения в фасаде `Auth`:

```
Auth::extend('riak', function($app)
{
    return new RiakUserProvider($app['riak.connection']);
});
```

Когда вы зарегистрировали драйвер методом `extend` вы можете активировать него в вашем файле настроек `config/auth.php`.

Расширения посредством IoC

Почти каждый сервис-провайдер Laravel получает свои объекты из контейнера IoC. Вы можете увидеть список сервис-

провайдеров в вашем приложения в файле config/app.php. Вам стоит пробежаться по коду каждого из поставщиков в свободное время - сделав это вы получите намного более чёткое представление о том, какую именно функционалньость каждый из них добавляет к фреймворку, а также какие ключи используются для регистрации различных услуг в контейнере IoC.

Например, HashServiceProvider использует ключ hash для получения экземпляра Illuminate\Hashing\BcryptHasher из контейнера IoC. Вы можете легко расширить и перекрыть этот класс в вашем приложении, перекрыв эту привязку. Например:

```
<?php namespace App\Providers;

class SnappyHashProvider extends \Illuminate\Hashing\HashServiceProvider {

    public function boot()
    {
        $this->app->bindShared('hash', function()
        {
            return new \Snappy\Hashing\ScryptHasher;
        });

        parent::boot();
    }

}
```

Заметьте, что этот класс расширяет HashServiceProvider, а не базовый класс ServiceProvider. Как только вы расширили этот сервис-провайдер, измените HashServiceProvider в файле настроек config/app.php на имя вашего нового сервис-провайдера.

Это общий подход к расширению любого класса ядра, который привязан к IoC-контейнеру. Фактически каждый класс так или иначе привязан к нему, и с помощью вышеописанного метода может быть перекрыт. Опять же, посмотрев код сервис-провайдеров фреймворка, вы познакомитесь с тем, где различные классы привязываются к IoC-контейнеру и какие ключи для этого используются. Это отличный способ понять глубинную работу Laravel.

Введение

Laravel Elixir предназначен для сборки файлов вашего фронтэнда в css- и js-файлы, а также для выполнения различных задач, построенных на слежении за изменениями в файлах вашего проекта - например, запуска тестов. Laravel Elixir представляет собой гибкий и простой API для Gulp.

Установка и настройка

Установка node.js

Сначала убедитесь, стоит ли у вас node.js:

```
node -v
```

По умолчанию node.js есть в Laravel Homestead. Если вы не используете его, вы можете установить её вручную.

Gulp

Далее, вам нужно глобально установить Gulp.

```
npm install --global gulp
```

Laravel Elixir

Осталось установить собственно Elixir. В корне папки фреймворка вы можете видеть файл `package.json`. Он похож на `composer.json`, только предназначен не для Composer, а для пакетного менеджера node.js , который называется `npm`. Вы можете установить зависимости, заданные в этом файле одной командой:

```
npm install
```

Использование

Команды elixir записываются в файле `gulpfile.js`.

Компиляция Less

```
elixir(function(mix) {
    mix.less("app.less");
});
```

В данном примере подразумевается, что ваши less-файлы находятся в `resources/assets/less`.

Компиляция Sass

```
elixir(function(mix) {
    mix.sass("app.sass");
});
```

Подразумевается, что ваши sass-файлы находятся в `resources/assets/sass`.

Компиляция CoffeeScript

```
elixir(function(mix) {
    mix.coffee();
});
```

Подразумевается, что ваши CoffeeScript-файлы находятся в `resources/assets/coffee`.

Компиляция всех Less и CoffeeScript файлов

```
elixir(function(mix) {
    mix.less()
       .coffee();
});
```

Запуск PHPUnit тестов

```
elixir(function(mix) {
    mix.phpUnit();
});
```

Запуск PHPSpec тестов

```
elixir(function(mix) {
    mix.phpSpec();
});
```

Объединение Stylesheets

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ]);
});
```

Пути задаются относительно директории `resources/css`.

Объединение Stylesheets и сохранение в определённое место

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css');
});
```

Объединение Stylesheets в заданном каталоге

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css', 'public/css');
});
```

Третий аргумент в `styles` и `scripts` определяет папку, относительно которой будут искаться заданные файлы.

Объединение всех css в папке

```
elixir(function(mix) {
    mix.stylesIn("public/css");
```

```
});
```

Объединение javascript

```
elixir(function(mix) {
    mix.scripts([
        "jquery.js",
        "app.js"
    ]);
});
```

Как и в случае с css, пути задаются относительно папки `resources/js`

Объединение всех js в папке

```
elixir(function(mix) {
    mix.scriptsIn("public/js/some/directory");
});
```

Объединение нескольких наборов js

```
elixir(function(mix) {
    mix.scripts(['jquery.js', 'main.js'], 'public/js/main.js')
       .scripts(['forum.js', 'threads.js'], 'public/js/forum.js');
});
```

Добавление версии файла

```
elixir(function(mix) {
    mix.version("css/all.css");
});
```

Файл будет сохранён с уникальным именем (к имени файла добавляется хэш содержимого и получается что-то вроде `all-16d570a7.css`), чтобы исключить кэширование его на клиенте.

Внутри ваших шаблонов вы можете использовать хэлпер `elixir()` для указания урла для такого файла:

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
```

Этот хэлпер считает хэш указанного файла и добавляет его в урл. Все происходит автоматически !

Копировать файл в новое место

```
elixir(function(mix) {
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');
});
```

Копировать каталог в новое место

```
elixir(function(mix) {
    mix.copy('vendor/package/views', 'resources/views');
});
```

Соединение методов

Вы можете образовывать цепочки из методов:

```
elixir(function(mix) {
    mix.less("app.less")
       .coffee()
       .phpUnit()
       .version("css/bootstrap.css");
});
```

Gulp

Для выполнения зарегистрированных команд нужно запустить в командной строке `gulp`

Выполнение всех зарегистрированных команд

```
gulp
```

Запуск команд при изменении файлов

```
gulp watch
```

Запуск тестов при изменении классов

```
gulp tdd
```

> **Note:** По умолчанию подразумевается, что конамды выполняются в development-окружении и собираемые скрипты не минифицируются. Чтобы добавить минификацию запустите `gulp --production`.

Расширение

Вы можете создавать свои gulp-задачи и добавлять в elixir. Например, сделаем шуточную задачу, которая выводит в терминал некое сообщение:

```
var gulp = require("gulp");
var shell = require("gulp-shell");
var elixir = require("laravel-elixir");

elixir.extend("message", function(message) {

    gulp.task("say", function() {
        gulp.src("").pipe(shell("say " + message));
    });

    return this.queueTask("say");

});
```

Первый аргумент в `extend` - имя задачи, которое мы будет далее использовать в нашем gulpfile.js , а второй - функция-замыкание с собственно кодом задачи, написанной для gulp.

Вы также можете мониторить изменения в заданных файлах:

```
this.registerWatcher("message", "**/*.php");
```

Когда файл с путём, удовлетворяющем регекспу `**/*.php`, будет изменён - запустится задача `message`.

That's it! You may either place this at the top of your Gulpfile, or instead extract it to a custom tasks file. If you choose the latter approach, simply require it into your Gulpfile, like so:

Вы можете разместить этот код в верхней части вашего `gulpfile.js` , или в сыойм файле, и подключить его в

`gulpfile.js` следующей конструкцией:

```
require("./custom-tasks")
```

Дальше вы можете добавлять свою команду в микс:

```
elixir(function(mix) {
    mix.message("Tea, Earl Grey, Hot");
});
```

Теперь, как только gulp исполнит какую-либо задачу, в терминал выведется строка "Tea, Earl Grey, Hot".

Введение

Laravel предоставляет удобный механизм для использования стойкого шифрования алгоритмом AES на основе PHP-модуля Mcrypt.

Основы использования

Шифрование

```
$encrypted = Crypt::encrypt('secret');
```

> **Примечание:** Обязательно укажите строку из случайных символов длиной в 16, 24, или 32 символа в параметре `key` файла `config/app.php`. В противном случае, зашифрованное значение будет не очень стойким к взлому.

Дешифровка

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Настройка алгоритма шифрования и режима работы

Вы можете указать алгоритм шифрования и режим работы:

```
Crypt::setMode('cfb');

Crypt::setCipher($cipher);
```

Конфигурация

Возможности логгирования для вашего приложения описаны в классе `Illuminate\Foundation\Bootstrap\ConfigureLogging`. Это класс использует параметр `log` из файла `config/app.php`.

По умолчанию, логгер настроен на ежедневную смену файла, но вы можете изменить это поведение. Так как Laravel использует для логгирования популярную библиотеку Monolog, то вы можете использовать большое количество существующих обработчиков, которые предлагает Monolog.

Например, если вы хотите, чтобы все логи писались в один файл вместо ежедневной смены файла, вы можете изменить файл `config/app.php` так:

```
'log' => 'single'
```

«Из коробки» Laravel поддерживает три режима логгирования: `single`, `daily`, and `syslog`, но вы можете настроить механизм логгирования так, как вам нужно путём переопределения класса `ConfigureLogging`.

Детализация ошибок

Подробная детализация ошибок вашего приложения, отображаемая в браузере, контролируется параметром `app.debug` в файле `config/app.php` По умолчанию этот параметр определяется переменной среды окружения `APP_DEBUG`, которая установлена в файле `.env`.

Во время разработки на локальной машине рекомендуется установить значение переменной `APP_DEBUG` в `true`.

> **Примечание** Настоятельно рекомендуется отключать детализацю ошибок для рабочей среды выполнения.

Обработка ошибок

Все исключения обрабатываются классом `App\Exceptions\Handler`, который содержит два метода: `report` и `render`

Метод `report` используется для логгирования исключений в файл или отправки информации на сторонний сервис логгирования типа BugSnag. По умолчанию, метод `report` просто передаёт исключение своему базовому классу, который его логгирует, однако вы можете изменить это поведение. Если необходимо обрабатывать разные типы исключений разными путями, то можно использовать оператор PHP `instanceof`:

```
/**
 * Report or log an exception.
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param  \Exception  $e
 * @return void
 */
public function report(Exception $e)
{
    if ($e instanceof CustomException)
    {
        //
    }

    return parent::report($e);
}
```

Метод `render` служит для преобразования исключения в ответ, который можно отправить в браузер. По умолчанию, исключение просто передаётся в базовый класс, но вы можете изменить это поведение, например, проверять тип исключения для генерации разных ответов.

Свойство `dontReport` обработчика исключения содержит массив типов исключений, которые не будут логгироваться. По умолчанию, исключения, выброшенные в результате ошибки 404 не логгируются. Вы можете добавить и другие типы исключений в этот массив.

HTTP-исключения

Такие ошибки могут возникнуть во время обработки запроса от клиента. Это может быть ошибка "Страница не

найдена" (http-код 404), "Требуется авторизация" (401) или даже "Ошибка сервера" (500). Для того, чтобы отправить такой ответ, используйте следующее:

```
abort(404);
```

Опционально, вы можете установить свой ответ для возврата в браузер:

```
abort(403, 'Unauthorized action.');
```

Эти исключения могут быть возбуждены на любом этапе обработки запроса.

Изменение страницы 404

Для изменения страницы, выдаваемой при ошибке 404, необходимо создать файл `resources/views/errors/404.blade.php`.

Логирование

Стандартный механизм логирования представляет собой простую надстройку над мощной системой Monolog. По умолчанию Laravel настроен так, чтобы создавать новый лог-файл каждый день в каталоге `storage/logs`. Вы можете записывать в лог информацию таким образом:

```
Log::info('Вот кое-какая полезная информация.');

Log::warning('Что-то может идти не так.');

Log::error('Что-то действительно идёт не так.');
```

Журнал предоставляет 7 уровней критичности, определённые в RFC 5424: **debug**, **info**, **notice**, **warning**, **error**, **critical** и **alert**.

В метод записи можно передать массив данных о текущем состоянии:

```
Log::info('Log message', ['context' => 'Другая полезная информация.']);
```

Monolog имеет множество других методов, которые вам могут пригодиться. Если нужно, вы можете получить экземпляр его класса:

```
$monolog = Log::getMonolog();
```

Вы также можете зарегистрировать обработчик событий для отслеживания всех новых сообщений:

Отслеживание новых сообщений в логе

```
Log::listen(function($level, $message, $context)
{
    //
});
```

Основы использования

Events в Laravel - это простая реализация паттерна Observer, которая позволяет вам подписываться на так называемые события, которые генерируются в некотором месте вашего приложения и обрабатываются как правило в этом же самом запросе. Классы событий по умолчанию находятся в папке `app/Events`, а классы обработчиков событий - в `app/Handlers/Events`.

Создать класс события можно artisan-командой `make:event`:

```
php artisan make:event PodcastWasPurchased
```

Подписка на события

Сервис-провайдер `EventServiceProvider` - это удобное место для регистрации классов слушателей событий. В массиве `listen` перечисляются названия события (в ключе массива) и название класса, который его обрабатывает. Например, для нашего `PodcastWasPurchased`:

```
/**
 * The event handler mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Handlers\Events\EmailPurchaseConfirmation@handle',
    ],
];
```

Для генерации исполнителя (handler) события используйте artisan-команду `handler:event`:

```
php artisan handler:event EmailPurchaseConfirmation --event=PodcastWasPurchased
```

Но вообще, использование двух команд, `make:event` и `handler:event` каждый раз, когда вам нужно создать обработчик события - это неудобно. Вместо этого можно заполнить массив `listen` в `EventServiceProvider` и выполнить artisan-команду `event:generate`. Эта команда анализирует `listen` и создает все необходимые классы событий и обработчиков событий:

```
php artisan event:generate
```

Запуск события

Событие может быть запущено (fire) при помощи фасада `Event`:

```
$response = Event::fire(new PodcastWasPurchased($podcast));
```

Метод `fire` возвращает массив ответов (responses).

Вместо фасада можно использовать хэлпер:

```
event(new PodcastWasPurchased($podcast));
```

Слушатели в функциях-замыканиях

Чтобы упростить себе работу и не создавать два класса, вы можете создать слушателя в виде функции-замыкания. Сделать это можно в методе boot сервис-провайдера `EventServiceProvider`:

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // Обработка события...
});
```

Остановка распространения события

Иногда вам нужно остановить распространение события, чтобы до других обработчиков, подписанных на это событие, оно не дошло. Для этого надо вернуть `false` в обработчике события:

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // Обработка события...
```

```
    return false;
});
```

Обработка события в фоне

Хотите запустить обработчик события в фоне при помощи queue ? Это делается очень легко. При создании класса обработчика события добавьте флаг `--queued`:

```
php artisan handler:event SendPurchaseConfirmation --event=PodcastWasPurchased --queued
```

Или вручную поставьте реализацию (implements) интерфейса `Illuminate\Contracts\Queue\ShouldBeQueued`. Это всё ! Теперь при возникновении события фреймворк поместит класс обработки этого события в очередь, откуда он будет запущен диспетчером очереди.

Если во время выполнения метода `handle` не будет брошено ни одного непойманного исключения, то обработчик события по завершении сам удалится из очереди. Но если вы хотите иметь доступ к методам задачи в очереди `delete` and `release` (вернуть задачу в очередь), то добавьте в класс обработчика события трейт `Illuminate\Queue\InteractsWithQueue`.

```
public function handle(PodcastWasPurchased $event)
{
    if (true)
    {
        $this->release(30);
    }
}
```

Подписчики

Defining An Event Subscriber

Подписчики на события (Event Subscribers) - классы, которые могут быть подписаны на несколько событий и содержать сразу несколько обработчиков событий. Такой класс должен иметь метод `subscribe`, который принимает в аргументах инстанс диспетчера событий:

```
class UserEventHandler {

    /**
     * Handle user login events.
     */
    public function onUserLogin($event)
    {
        //
    }

    /**
     * Handle user logout events.
     */
    public function onUserLogout($event)
    {
        //
    }

    /**
     * Register the listeners for the subscriber.
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen('App\Events\UserLoggedIn', 'UserEventHandler@onUserLogin');

        $events->listen('App\Events\UserLoggedOut', 'UserEventHandler@onUserLogout');
    }

}
```

Регистрация Event Subscriber

После того как класс определён, его можно зарегистрировать следующим образом:

```
$subscriber = new UserEventHandler;

Event::subscribe($subscriber);
```

Вы также можете использовать сервис-контейнер для того, чтобы получить объект своего подписчика на события:

```
Event::subscribe('UserEventHandler');
```

Введение

Laravel предоставляет чудесную абстракцию для работы с файловой системой - Flysystem от Frank de Jonge. У Flysystem есть драйвера для работы с Amazon S3, Rackspace Cloud Storage, и, конечно, с локальной файловой системой. Теперь невероятно просто переключиться с хранения файлов на сервере на хранение файлов на S3!

Настройка

Настройки Flysystem находятся в файле `config/filesystems.php`. Внутри него вы можете настроить несколько `'disks'`. Каждый диск представляет свой тип хранения - локальная файловая система или облачные хранилища. В файле настроек уже есть примеры конфигурации дисков для каждого из поддерживаемых хранилищ.

Перед использованием S3 или Rackspace вы должны установить при помощи Composer соответствующие пакеты:

- Amazon S3: `"league/flysystem-aws-s3-v2": "~1.0"`
- Rackspace: `"league/flysystem-rackspace": "~1.0"`

Вы можете сконфигурировать несколько дисков, с одним и тем же драйвером.

Обратите внимание, что когда вы используете драйвер `'local'`, все пути в командах будут считаться от пути, заданного в параметре `'root'`. По умолчанию это папка `storage/app`. Например, этот код создаст файл `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Основы использования

Для взаимодействия с вашими дисками вы можете использовать фасад `Storage` или внедрить в конструктор класса объект, реализующий `Illuminate\Contracts\Filesystem\Factory`, используя сервис-контейнер.

Подключение диска

```
$disk = Storage::disk('s3');

$disk = Storage::disk('local');
```

Определение, существует ли файл

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

Выполнение метода на дефолтном диске

```
if (Storage::exists('file.jpg'))
{
    //
}
```

Чтение файла

```
$contents = Storage::get('file.jpg');
```

Запись в файл

```
Storage::put('file.jpg', $contents);
```

Добавление контента в начало файла

```
Storage::prepend('file.log', 'Prepended Text');
```

Добавление контента в конец файла

```
Storage::append('file.log', 'Appended Text');
```

Удвление файла

```
Storage::delete('file.jpg');
```

```
Storage::delete(['file1.jpg', 'file2.jpg']);
```

Копировать файл

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

Переместить файл

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Получить размер файла

```
$size = Storage::size('file1.jpg');
```

Получить время последней модификации файла (UNIX)

```
$time = Storage::lastModified('file1.jpg');
```

Получить все файлы в директории

```
$files = Storage::files($directory);

// И из всех поддиректорий - рекурсивно...
$files = Storage::allFiles($directory);
```

Получить все поддиректори

```
$directories = Storage::directories($directory);

// рекурсивно...
$directories = Storage::allDirectories($directory);
```

Создать директорию

```
Storage::makeDirectory($directory);
```

Удалить директорию

```
Storage::deleteDirectory($directory);
```

Введение

Фасад Hash предоставляет механизм для защищённого хэширования паролей алгоритмом Bcrypt. Если вы используете контроллер AuthController, идущий «из коробки», то он уже позаботился о сверке сохранённого хэша пароля и пароля, введённого пользователем, используя для этого трейт AuthenticatesAndRegistersUsers и сервис Registrar.

Основы использования

Хэширование пароля

```
$password = Hash::make('secret');
```

Так же можно использовать функцию-хелпер bcrypt:

```
$password = bcrypt('secret');
```

Сверка пароля и хэша

```
if (Hash::check('secret', $hashedPassword))
{
    // Пароль верен...
}
```

Провера не необходимость рехэша пароля

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

Массивы

array_add

Добавить указанную пару ключ/значение в массив, если она там ещё не существует.

```
$array = array('foo' => 'bar');

$array = array_add($array, 'key', 'value');
```

array_divide

Вернуть два массива - один с ключами, другой со значениями оригинального массива.

```
$array = array('foo' => 'bar');

list($keys, $values) = array_divide($array);
```

array_dot

Сделать многоуровневый массив одноуровневым, объединяя вложенные массивы с помощью точки в именах.

```
$array = array('foo' => array('bar' => 'baz'));

$array = array_dot($array);

// array('foo.bar' => 'baz');
```

array_except

Удалить указанную пару ключ/значение из массива.

```
$array = array_except($array, array('ключи', 'для', 'удаления'));
```

array_fetch

Вернуть одноуровневый массив с выбранными элементами по переданному пути.

```
$array = array(
    array('developer' => array('name' => 'Taylor')),
    array('developer' => array('name' => 'Dayle')),
);

$array = array_fetch($array, 'developer.name');

// array('Taylor', 'Dayle');
```

array_first

Вернуть первый элемент массива, прошедший (return true) требуемый тест.

```
$array = array(100, 200, 300);

$value = array_first($array, function($key, $value)
{
    return $value >= 150;
});
```

Третьим параметром можно передать значение по умолчанию:

```
$value = array_first($array, $callback, $default);
```

array_flatten

Сделать многоуровневый массив плоским.

```
$array = array('name' => 'Joe', 'languages' => array('PHP', 'Ruby'));
```

```
$array = array_flatten($array);

// array('Joe', 'PHP', 'Ruby');
```

array_forget

Удалить указанную пару ключ/значение из многоуровневого массива, используя синтаксис имени с точкой.

```
$array = array('names' => array('joe' => array('programmer')));

array_forget($array, 'names.joe');
```

array_get

Вернуть значение из многоуровневого массива, используя синтаксис имени с точкой.

```
$array = array('names' => array('joe' => array('programmer')));

$value = array_get($array, 'names.joe');

$value = array_get($array, 'names.john', 'default');
```

> **Примечание:** Нужен `array_get` для объектов? Используйте `object_get`.

array_only

Вернуть из массива только указанные пары ключ/значения.

```
$array = array('name' => 'Joe', 'age' => 27, 'votes' => 1);

$array = array_only($array, array('name', 'votes'));
```

array_pluck

Извлечь значения из многоуровневого массива, соответствующие переданному ключу.

```
$array = array(array('name' => 'Taylor'), array('name' => 'Dayle'));

$array = array_pluck($array, 'name');

// array('Taylor', 'Dayle');
```

array_pull

Извлечь значения из многоуровневого массива, соответствующие переданному ключу, и удалить их.

```
$array = array('name' => 'Taylor', 'age' => 27);

$name = array_pull($array, 'name');
```

array_set

Установить значение в многоуровневом массиве, используя синтаксис имени с точкой.

```
$array = array('names' => array('programmer' => 'Joe'));

array_set($array, 'names.editor', 'Taylor');
```

array_sort

Отсортировать массив по результатам вызывов переданной функции-замыкания.

```
$array = array(
    array('name' => 'Jill'),
    array('name' => 'Barry'),
);

$array = array_values(array_sort($array, function($value)
{
    return $value['name'];
}));
```

array_where

Отфильтровать массив функцией-замыканием.

```
$array = array(100, '200', 300, '400', 500);

$array = array_where($array, function($key, $value)
{
    return is_string($value);
});

// Array ( [1] => 200 [3] => 400 )
```

head

Вернуть первый элемент массива. Полезно при сцеплении методов в PHP 5.3.x.

```
$first = head($this->returnsArray('foo'));
```

last

Вернуть последний элемент массива. Полезно при сцеплении методов.

```
$last = last($this->returnsArray('foo'));
```

Пути

app_path

Получить абсолютный путь к папке `app`.

base_path

Получить абсолютный путь к корневой папке приложения.

public_path

Получить абсолютный путь к папке `public`.

storage_path

Получить абсолютный путь к папке `storage`.

Строки

camel_case

Преобразовать строку к `camelCase`.

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename

Получить имя класса переданного класса без пространства имён.

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e

Выполнить над строкой `htmlentities` в кодировке UTF-8.

```
$entities = e('<html>foo</html>');
```

ends_with

Определить, заказчивается ли строка переданной подстрокой.

```
$value = ends_with('This is my name', 'name');
```

snake_case

Преобразовать строку к snake_case (стиль именования Си, с подчёркиваниями вместо пробелов - прим. пер.).

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit

Ограничить строку заданным количеством символов и символами окончания.

```
str_limit($value, $limit = 100, $end = '...')
```

Example:

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with

Определить, начинается ли строка с переданной подстроки.

```
$value = starts_with('This is my name', 'This');
```

str_contains

Определить, содержит ли строка переданную подстрокую.

```
$value = str_contains('This is my name', 'my');
```

str_finish

Добавить одно вхождение подстроки в конец переданной строки и удалить повторы в конце, если они есть.

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is

Определить, соответствует ли строка маске. Можно использовать звёздочки (*) как символы подстановки.

```
$value = str_is('foo*', 'foobar');
```

str_plural

Преобразовать слово-строку во множественное число (только для английского языка).

```
$plural = str_plural('car');
```

str_random

Создать последовательность случайных символов заданной длины.

```
$string = str_random(40);
```

str_singular

Преобразовать слово-строку в единственное число (только для английского языка).

```
$singular = str_singular('cars');
```

str_slug

Создать строку-идентификатор (slug) с учётом ограничений URL.

```
str_slug($title, $separator);
```

Пример:

```
$title = str_slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

studly_case

Преобразовать строку в StudlyCase.

```
$value = studly_case('foo_bar');

// FooBar
```

trans

Перевести переданную языковую строку. Это алиас для Lang::get.

```
$value = trans('validation.required'):
```

trans_choice

Перевести переданную языковую строку с изменениями. Алиас для Lang::choice.

```
$value = trans_choice('foo.bar', $count);
```

URLs

action

Сгенерировать URL для заданного экшна (метода) контроллера.

```
$url = action('HomeController@getIndex', $params);
```

route

Сгенерировать URL для заданного именованного роута.

```
$url = route('routeName', $params);
```

asset

Сгенерировать URL ко внешнему ресурсу (изображению и пр.).

```
$url = asset('img/photo.jpg');
```

link_to

Сгенерировать HTML-ссылку на указанный URL.

```
echo link_to('foo/bar', $title, $attributes = array(), $secure = null);
```

link*to*asset

Сгенерировать HTML-ссылку на внешний ресурс (изображение и пр.).

```
echo link_to_asset('foo/bar.zip', $title, $attributes = array(), $secure = null);
```

link*to*route

Сгенерировать HTML-ссылку на заданный именованный маршрут.

```
echo link_to_route('route.name', $title, $parameters = array(), $attributes = array());
```

link*to*action

Сгенерировать HTML-ссылку на заданное действие контроллера.

```
echo link_to_action('HomeController@getIndex', $title, $parameters = array(), $attributes = array());
```

secure_asset

Сгенерировать HTML-ссылку на внешний ресурс (изображение и пр.) через HTTPS.

```
echo secure_asset('foo/bar.zip', $title, $attributes = array());
```

secure_url

Сгенерировать HTML-ссылку на указанный путь через HTTPS.

```
echo secure_url('foo/bar', $parameters = array());
```

url

Сгенерировать HTML-ссылку на указанный абсолютный путь.

```
echo url('foo/bar', $parameters = array(), $secure = null);
```

Прочее

csrf_token

Получить текущее значение CSRF-токена.

```
$token = csrf_token();
```

dd

Вывести дамп переменной и завершить выполнение скрипта.

```
dd($value);
```

value

Если переданное значение - функция-замыкание, вызвать её и вернуть результат. В противном случае вернуть само значение.

```
$value = value(function() { return 'bar'; });
```

with

Вернуть переданный объект. Полезно при сцеплении методов в PHP 5.3.x.

```
$value = with(new Foo)->doWork();
```

Введение

Фасад Lang даёт возможность удобного получения языковых строк, позволяя вашему приложению поддерживать несколько языков интерфейса.

Файлы локализации

Файлы локализации хранятся в папке `resource/lang` Внутри неё должны располагаться подпапки - языки, поддерживаемые приложением:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

Пример файла локализации

Файлы локализации возвращают массив пар ключ/значение:

```
<?php

return array(
    'welcome' => 'Добро пожаловать на мой сайт!'
);
```

Язык по умолчанию указан в файле настроек `config/app.php`. Вы можете изменить текущий язык во время работы вашего приложения методом `App::setLocale`:

```
App::setLocale('es');
```

Резервный язык локализации

Вы также можете установить резервный язык локализации - в случае, если для основного языка нет вариантов перевода, будет браться строка из резервного файла локализации. Обычно это английский язык, но вы можете это поменять. Настройка находится в файле `config/app.php`:

```
'fallback_locale' => 'en',
```

Основы использования

Получение строк из языкового файла

```
echo Lang::get('messages.welcome');
```

Первый компонент строки (до точки), передаваемый методу `get` - имя языкового файла, а после указывается имя строки, которую нужно получить.

> **Примечание:** если строка не найдена, то метод `get` вернёт её путь (ключ).

Вы также можете использовать функцию `trans` - короткий способ вызова метода `Lang::get`:

```
echo trans('messages.welcome');
```

Замена параметров внутри строк

Сперва определите параметр в языковой строке:

```
'welcome' => 'Welcome, :name',
```

Затем передайте массив вторым аргументом методу `Lang::get`:

```
echo Lang::get('messages.welcome', array('name' => 'Dayle'));
```

Проверка существования языковой строки

```
if (Lang::has('messages.welcome'))
{
    //
}
```

Формы множественного числа

Формы множественного числа - проблема для многих языков, так как все они имеют разные сложные правила для их получения. Однако вы можете легко справиться с ней в ваших языковых файлах используя символ «|» для разделения форм единственного и мнжественного чисел.

```
'apples' => 'There is one apple|There are many apples',
```

Для получения такой строки используется метод `Lang::choice`:

```
echo Lang::choice('messages.apples', 10);
```

Вы можете указать не два, а несколько вариантов выражения множественного числа:

```
echo Lang::choice('товар|товара|товаров', $count, array(), 'ru');
```

Благодаря тому, что Laravel использует компонент Symfony Translation вы можете легко создать более точные правила для проверки числа:

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

Сообщения валидации

О том, как использовать файлы локализации для сообщений валидации, смотрите соответствующий раздел документации.

Перекрытие файлов локализации из пакетов

Многие пакеты идут со своими файлами локализации. Вы можете «перекрыть» их, располагая файлы в папках `resources/lang/packages/{locale}/{package}`. Например, если вам надо перекрыть файл `messages.php` пакета `skyrim/hearthfire`, путь до вашего файла локализации должен выглядеть так: `resources/lang/packages/en/hearthfire/messages.php`. Нет нужды дублировать файл целиком, можно указать только те ключи, которые должны быть перекрыты.

Настройка

Laravel предоставляет простой интерфейс к популярной библиотеке SwiftMailer. Главный файл настроек - `app/config/mail.php` - содержит всевозможные параметры, позволяющие вам менять SMTP-сервер, порт, логин, пароль, а также устанавливать глобальный адрес `from` для исходящих сообщений. Вы можете использовать любой SMTP-сервер, либо стандартную функцию PHP `mail` - для этого установите параметр `driver` в значение `mail`. Кроме того, доступен драйвер `sendmail`.

Отправка через API

Laravel содержит драйвера отправки почты через HTTP API сервисов Mailgun и Mandrill. Отправка через API как правило работает быстрее, чем отправка через протокол SMTP. Оба этих драйвера требуют наличия пакета Guzzle для осуществления HTTP-запросов. Чтобы включить его в свое приложение, добавьте в `composer.json`

```
"guzzlehttp/guzzle": "~4.0"
```

и выполните

```
composer update
```

Драйвер Mailgun

В файле `app/config/mail.php` установите опцию `driver` в 'mailgun'. Создайте файл `app/config/services.php`, в котором укажите данные вашего аккаунта на mailgun.com :

```
'mailgun' => array(
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
),
```

Драйвер Mandrill

В файле `app/config/mail.php` установите опцию `driver` в 'mandrill'. Создайте файл `app/config/services.php`, в котором укажите данные вашего аккаунта на mandrill.com :

```
'mandrill' => array(
    'secret' => 'your-mandrill-key',
),
```

Log Driver

Если в файле `app/config/mail.php` установить опцию `driver` в 'log', то все отправляемые письма будут записываться в лог-файл фреймворка и не будут рассылаться. Этот вариант используется для отладки.

Основы использования

Метод `Mail::send` используется для отправки сообщения:

```
Mail::send('emails.welcome', array('key' => 'value'), function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Первый параметр - имя шаблона, который должен использоваться для текста сообщения. Второй - ассоциативный массив переменных, передаваемых в шаблон. Третий - функция-замыкание, позволяющая вам внести дополнительные настройки в сообщение.

> **Примечание:** Переменная `$message` всегда передаётся в ваш шаблон и позволяет вам прикреплять вложения. Таким образом, вам не стоит передавать одноимённую переменную в массиве `$data`.

В дополнение к шаблону в формате HTML вы можете указать текстовый шаблон письма:

```
Mail::send(array('html.view', 'text.view'), $data, $callback);
```

Вы также можете оставить только один формат, передав массив с ключом `html` или `text`:

```
Mail::send(array('text' => 'view'), $data, $callback);
```

Вы можете указывать другие настройки для сообщения, например, копии или вложения:

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');

    $message->attach($pathToFile);
});
```

При добавлении файлов можно указывать их MIME-тип и/или отображаемое имя:

```
$message->attach($pathToFile, array('as' => $display, 'mime' => $mime));
```

Если вам надо просто отправить письмом несколько слов, то вместо того, чтобы создавать для этого шаблон, воспользуйтесь простым методом `raw`:

```
Mail::raw('Текст письма', function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');
});
```

> **Примечание:** Объект `$message`, передаваемый функции-замыканию метода `Mail::send`, наследует класс собщения SwiftMailer, что позволяет вам вызывать любые методы для создания своего сообщения.

Добавление встроенных вложений

Обычно добавление встроенных вложений в письмо обычно утомительное занятие, однако Laravel делает его проще, позволяя вам добавлять файлы и получать соответствующие CID.

> Встроенные (inline) вложения - файлы, не видимые получателю в списке вложений, но используемые внутри HTML-тела сообщения; CID - уникальный идентификатор внутри данного сообщения, используемый вместо URL в таких атрибутах, как `src` - прим. пер.

Добавление картинки в шаблон сообщения

```
<body>
    Вот какая-то картинка:

    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

Добавление встроенной в html картинки (data:image)

```
<body>
    А вот картинка, полученная из строки с данными:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

Переменная `$message` всегда передаётся шаблонам сообщений классом `Mail`.

Очереди отправки

Из-за того, что отправка писем может сильно повлиять на время отклика приложения, многие разработчики помещают их в фоновую очередь на отправку. Laravel позволяет делать это, используя единое API очередей. Для помещения сообщения в очередь просто используйте метод `Mail::queue()`:

Помещение сообщения в очередь отправки

```
Mail::queue('emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Вы можете задержать отправку сообщения на нужное число секунд методом `later`:

```
Mail::later(5, 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Если же вы хотите поместить сообщение в определённую очередь отправки, то используйте методы `queueOn` и `laterOn`:

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Локальная разработка

При разработке приложения обычно предпочтительно отключить доставку отправляемых сообщений. Для этого вы можете либо вызывать метод `Mail::pretend`, либо установить параметр `pretend` в значение `true` в файле настроек `config/mail.php`. Когда это сделано, сообщения будут записываться в файл журнала вашего приложения, вместо того, чтобы быть отправленными получателю.

Если вы хотите отладить вид отсылаемых писем, воспользуйтесь сервисом типа MailTrap.

Введение

Пакеты (packages) - основной способ добавления нового функционала в Laravel. Пакеты могут быть всем, чем угодно - от классов для удобной работы с датами типа Carbon до целых библиотек BDD-тестирования наподобие Behat.

Конечно, всё это разные типы пакетов. Некоторые пакеты самостоятельны, что позволяет им работать в составе любого фреймворка, а не только Laravel. Примерами таких отдельных пакетов являются Carbon и Behat. Любой из них может быть использован в Laravel после простого указания его в файле `composer.json`.

С другой стороны, некоторые пакеты разработаны специально для использования в Laravel. Они могли содержать роуты, контроллеры, шаблоны, настройки и миграции. Так как для разработки самостоятельных пакетов нет особенных правил, этот раздел документации в основном посвящён разработке именно пакетов для Laravel.

Все пакеты Laravel распространяются через Packagist и Composer, поэтому нужно изучить эти прекрасные средства распространения кода для PHP.

Файлы шаблонов (views)

Структура вашего пакета зависит от вас. Обычно пакет имеет в составе один или несколько сервис-провайдеров. Сервис-провайдеры содержат регистрацию классов в IoC и определяют, где находятся конфиги, шаблоны (views) и локализационные файлы пакета.

Шаблоны

К шаблонам пакета можно обращаться через так называемую неймспейс-конструкцию `::`:

```
return view('package::view.name');
```

Вам нужно зарегистрить неймспейс шаблонов, указав, где находится папка шаблонов пакета. Делается это в методе `boot()` сервис-провайдера. Например, так регистрируется неймспейс `courier`

```
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

Теперь вы можете использовать в контроллере или роуте шаблон пакета таким образом:

```
return view('courier::view.name');
```

Когда вы регистрируете папку в `loadViewsFrom`, Laravel на самом деле регистрирует **две** папки - одну, которую вы указали и другую - `resources/views/vendor`. В этой папке лежат шаблоны пакетов, которые пользователи могут изменять под себя. Когда запрашивается шаблон `courier::view.name`, Laravel сначала пытается загрузить этот шаблон из `resources/views/vendor/courier`, а затем из папки, заданной в `loadViewsFrom`.

Публикация шаблонов для изменения

Чтобы разрешить копировать шаблоны пакета в папку `resources/views/vendor`, где пользователи могут их изменять, вы должны использовать метод `publishes` в `boot` сервис-провайдера пакета:

```
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/views/vendor/courier'),
    ]);
}
```

Теперь, когда пользователь исполнит команду `vendor:publish`, шаблоны пакета будут скопированы в соответствующу папку.

Чтобы перезаписать имеющися файлы используйте флаг `--force`:

```
php artisan vendor:publish --force
```

> **Примечание:** Вы можете использовать publishes не только для шаблонов, но и для **любых других файлов**, которые хотите скопировать из пакета в основное приложение пользователя.

Файлы переводов

Файлы локализации пакета доступны при помощи неймспейс-доступа `::`:

```
return trans('package::file.line');
```

Чтобы зарегистрировать этот неймспейс, воспользуйтесь методом `loadTranslationsFrom` в `boot` сервис-провайдера пакета. Например, для неймспейса `courier`:

```
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

Папка `translations` пакета должна содержать языковые подпапки, такие как `en`, `es`, `ru`, etc.

Файлы конфигов

Как правило, при установке пакета необходимо, чтобы пользователь мог перенести файлы конфигов к себе в приложение, чтобы иметь возможность менять там настройки. Для организации этой возможности воспользуйтесь методом `publishes` в `boot` сервис-провайдера пакета:

```
$this->publishes([
    __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
]);
```

Теперь когда пользователь после установки вашего пакета выполнит artisan-команду `vendor:publish`, указанный файл будет скопирован в новое место, а именно в папку конфигов.

Доступ к конфигу осуществляется штатным образом, без неймспейс-нотации:

```
$value = config('courier.option');
```

По умолчанию Laravel будет брать конфиг из приложения пользователя. Но вы можете сделать так, что конфиг в области приложения и конфиг в области пакета будут объединены - например, если хотите разрешить пользователям указывать в скопированном конфиге не все опции, а только некоторые, которые они хотят изменить. Для этого воспользуйтесь методом `mergeConfigFrom` в методе `register` сервис-провайдера пакета:

```
$this->mergeConfigFrom(
    __DIR__.'/path/to/config/courier.php', 'courier'
);
```

Публикация групп файлов

ВЫ можете разделить файлы, которые можно публиковать в приложение, на группы, и публиковать их раздельно. Например, в пакете есть конфиг и миграции:

```
// Публикация конфига
$this->publishes([
    __DIR__.'/../config/package.php', config_path('package.php')
], 'config');

// Публикация миграций
$this->publishes([
    __DIR__.'/../database/migrations/' => base_path('/database/migrations')
], 'migrations');
```

Тепрь пользователь может опубликовать у себя, например, только конфиг:

```
php artisan vendor:publish --provider="Vendor\Providers\PackageServiceProvider" --tag="config"
```

Файлы маршрутов

Чтобы добавить в приложение маршруты пакета, подключите файл с ними в методе `boot` сервис-провайдера пакета:

Вставка роутов в сервис-провайдер

```
public function boot()
{
```

```
    include __DIR__.'/../../routes.php';
}
```

Note: Если ваш пакет использует контроллеры, проследите, чтобы путь до них был корректно задан в `composer.json` в `autoload` сессии.

Введение

В других фрейморках пагинация (постраничный вывод данных) может быть большой проблемой. Laravel же делает этот процесс безболезненным. Фреймворк способен сам генерировать диапазон ссылок относительно текущей страницы. Сгенерированная разметка совместима с фреймворком Twitter Bootstrap.

Использование

Есть несколько способов разделения данных на страницы. Самый простой - используя метод `paginate` объекта-построителя запросов или в связке с моделями Eloquent.

Постраничный вывод выборки из БД

```
$users = DB::table('users')->paginate(15);
```

> **Примечание:** Если вы используете `groupBy` в запросе, то встроенная пагинация Laravel будет работать неэффективно. В этом случае вам нужно создать пагинатор вручную.

Создание пагинатора вручную

Иногда возникает потребность создать пагинатор вручную, передав ему массив элементов. Сделать это можно, создав объект `Illuminate\Pagination\Paginator` или `Illuminate\Pagination\LengthAwarePaginator`, в зависимости от задачи.

Постраничный вывод моделей Eloquent

```
$users = User::where('votes', '>', 100)->paginate(15);
```

Аргумент, передаваемый методу `paginate` - число строк, которые вы хотите видеть на одной странице. Блок пагинации в шаблоне отображаются методом `render`:

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->render(); ?>
```

Это всё, что нужно для создания страничного вывода! Заметьте, что нам не понадобилось уведомлять фреймворк о номере текущей страницы - Laravel определит его сам. Номер страницы добавляется к URL в виде параметра запроса: `?page=N`.

Вы можете получить информацию о текущем положении с помощью этих методов:

- `currentPage`
- `lastPage`
- `perPage`
- `total`
- `count`

«Упрощённая пагинация»

Если вам нужно выводить только ссылки «Следующая страница» и «Предыдущая страница», то вы можете использовать метод `simplePaginate`. В таком случае запрос в БД будет более простым. Это полезно на очень больших объемах данных и там, где пользователю нужны первые несколько страниц и нет необходимости переходить в самую глубину.

```
$someUsers = User::where('votes', '>', 100)->simplePaginate(15);
```

Настройка URL для вывода ссылок

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

$users->setPath('custom/url');
```

Пример выше создаст ссылки наподобие такой: `http://example.com/custom/url?page=2`

Параметры в ссылках

Вы можете добавить параметры запросов к ссылкам страниц с помощью метода appends страничного объекта:

```
<?php echo $users->appends(['sort' => 'votes'])->render(); ?>
```

Код выше создаст ссылки наподобие http://example.com/something?page=2&sort=votes

Чтобы добавить к URL хэш-параметр («#xyz»), используйте метод `fragment`:

```
<?php echo $users->fragment('foo')->render(); ?>
```

Код выше создаст ссылки типа http://example.com/something?page=2#foo

Конвертация в JSON

Класс `Paginator` реализует (implements) `Illuminate\Contracts\Support\JsonableInterface`, следовательно, у него есть метод `toJson`, который используется для вывода пагинируемой информации в формате json. Помимо пагинируемых данных, которые располагаются в `data`, этот метод добавляет мета-информацию, а именно: `total`, `current_page` и `last_page`.

Настройка

В Laravel компонент Queue предоставляет единое API для различных сервисов очередей. Очереди позволяют вам отложить выполнение времязатратной задачи, такой как отправка e-mail, на более позднее время, таким образом на порядок ускоряя загрузку (генерацию) страницы.

Настройки очередей хранятся в файле `config/queue.php`. В нём вы найдёте настройки для драйверов-связей, которые поставляются вместе с фреймворком: `database` - очередь, построенная на таблице в БД, Beanstalkd, IronMQ, Amazon SQS, `sync` - синхронный драйвер для локального использования и `null` - запрет использования очередей.

Создание таблицы очереди

Если вы используете драйвер `database`, то вам нужно создать таблицу, в которой будет организована очередь:

```
php artisan queue:table
```

Установка пакетов

Упомянутые выше драйвера имеют следующие зависимости:

- Amazon SQS: `aws/aws-sdk-php`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq`
- Redis: `predis/predis ~1.0`

Использование очередей

Добавление новой задачи в очередь

В очередь в качестве задач (jobs) могут быть добавлены так называемые команды, которые находятся в `App\Commands`. Вы можете создать такую команду следующим образом:

```
php artisan make:command SendEmail --queued
```

Чтобы добавить команду в очередь, воспользуйтесь методом `push`:

```
Queue::push(new SendEmail($message));
```

> **Примечание:** В этом примере мы используем фасад `Queue` для отправки задачи в очередь. Однако, лучшим способом было бы воспользоваться командной шиной и запустить команду, которая настроена во время своей работы уходить в очередь и исполняться там. Дальше по тексту мы продолжим использовать фасад `Queue`, однако настоятельно рекомендуем ознакомиться с командной шиной, так как этот концепт позволяет создавать команды, которые можно запускать универсально - и синхронно и в фоне, через очереди.

По умолчанию `make:command` создает "self-handling" команду, то есть содержащую метод и конструктор класса-команды и метод `handle`, который исполняет команду. Вы можете передать в этот метод необходимые для работы классы в виде аргументов:

```
public function handle(UserRepository $users)
{
    //
}
```

Если же вы хотите разделить команду на два класса, т.е. выделить выполнение в отдельный класс, то добавьте флаг `--handler` при создании:

```
php artisan make:command SendEmail --queued --handler
```

Хэндлер, т.е. класс с методом `hanlde` будет помещён в `App\Handlers\Commands`.

Добавление задачи в определенную очередь

У приложения может быть несколько очередей. Можно поместить команду в определенную очередь:

```
Queue::pushOn('emails', new SendEmail($message));
```

Передача данных нескольким задачам сразу

Если вам надо передать одни и те же данные нескольким задачам в очереди, вы можете использовать метод `Queue::bulk`:

```
Queue::bulk(array(new SendEmail($message), new AnotherCommand));
```

Отложенное выполнение задачи

Иногда вам нужно, чтобы задача начала исполняться не сразу после занесения её в очередь, а спустя какое-то время. Например, выслать пользователю письмо спустя 15 минут после регистрации. Для этого существует метод `Queue::later`:

```
$date = Carbon::now()->addMinutes(15);

Queue::later($date, new SendEmail($message));
```

Здесь для задания временного периода используется библиотека для работы с временем и датой Carbon, но `$date` может быть и просто целым числом секунд.

> **Примечание:** У Amazon SQS есть ограничение - максимальная пауза отложенного запуска составляет 900 секунд (15 минут).

Очереди и модели Eloquent

Если ваша команда, которая отправляется в очередь, принимает модель Eloquent в своём конструкторе, в очередь будет передан только её ID. Когда команда выполняется обработчиком очереди (вызывается метод `handle`), фреймворк автоматически загружает из базы данных экземпляр модели с данным ID. Это происходит полностью прозрачно для приложения.

Удаление выполненной задачи

Когда команда успешно заканчивает выполнение, т.е. в процессе её работы не было брошено ни одно непойманное исключение, она автоматически удаляется из очереди.

Вы можете вручную удалить команду из очереди и вернуть её обратно в очередь. Для этого вам надо добавить в команду трейт `Illuminate\Queue\InteractsWithQueue`, который предоставляет методы `delete` и `release`.

```
public function handle(SendEmail $command)
{
    if (true)
    {
        $this->release(30);
    }
}
```

Параметр в методе `release` - число секунд, через которое данная команда должна вернуться в очередь.

Помещение команды обратно в очередь

Если во время выполнения команды было брошено непойманное исключение, то команда автоматически помещается обратно в очередь. Так происходит до тех пор, пока не превысится максимальное число попыток, заданное в параметре `--tries` вашего слушателя очереди.

Получение числа попыток запуска

Вы можете узнать, сколько попыток перезапуска команды уже было:

```
if ($this->attempts() > 3)
{
    //
}
```

> **Примечание:** Вы должны включить в класс трейт `Illuminate\Queue\InteractsWithQueue` чтобы использовать этот метод.

Добавление функций-замыканий в очередь

Вы можете помещать в очередь и функции-замыкания. Это очень удобно для простых задач, которым нужно выполняться в очереди.

Добавить функцию-замыкание в очередь

```
Queue::push(function($job) use ($id)
{
    Account::delete($id);

    $job->delete();
});
```

> **Примечание:** Старайтесь не передавать модели в конструкции `use()`. Вместо этого передавайте ID объекта, и получайте его из БД в функции-замыкании.

При использовании push-очередей Iron.io, будьте особенно внимательны при добавлении замыканий. Конечная точка выполнения, получающая ваше сообщение из очереди, должна проверить входящую последовательность-ключ, чтобы удостовериться, что запрос действительно исходит от Iron.io. Например, ваша конечная push-точка может иметь адрес вида `https://yourapp.com/queue/receive?token=SecretToken` где значение `token` можно проверять перед собственно обработкой задачи.

Обработчик очереди

Задачи, помещенные в очередь должен кто-то исполнять. Laravel включает в себя Artisan-задачу, которая раз в несколько секунд опрашивает очередь и, если в очереди есть задача, запускает в отдельном процессе рабочего (worker), который выполняет её. Задачи запускаются не параллельно, а последовательно. Вы можете запустить её командой `queue:listen`:

Запуск сервера выполнения задач

```
php artisan queue:listen
```

Вы также можете указать, какое именно соединение должно прослушиваться:

```
php artisan queue:listen connection
```

Заметьте, что когда это задание запущено оно будет продолжать работать, пока вы не остановите его вручную. Вы можете использовать монитор процессов, такой как Supervisor, чтобы удостовериться, что задание продолжает работать.

Вы можете указать, задачи из каких очередей нужно будет исполнять в первую очередь. Для этого перечислите их через запятую в порядке уменьшения приоритета:

```
php artisan queue:listen --queue=high,low
```

Задачи из `high` будут всегда выполняться раньше задач из `low`.

Указание числа секунд для работы сервера

Вы можете указать число секунд, в течении которых будут выполняться задачи - например, для того, чтобы поставить `queue:listen` в cron на запуск раз в минуту.

```
php artisan queue:listen --timeout=60
```

Уменьшение частоты опроса очереди

Для уменьшения нагрузки на очередь, вы можете указать время, которое сервер выполнения задач должен бездействовать перед опросом очереди.

```
php artisan queue:listen --sleep=5
```

Если очередь пуста, она будет опрашиваться раз в 5 секунд. Если в очереди есть задачи, они исполняются без задержек.

Обработка только первой задачи в очереди

Для обработки только одной (первой) задачи можно использовать команду `queue:work`:

```
php artisan queue:work
```

Обработчик очереди в режиме демона

Команда `queue:listen` вызывает команды `queue:work`, которая, как и при HTTP-запросе браузера, инициализирует фреймворк, вызывает метод класса или функцию-замыкание, десериализованную из очереди, и затем завершает

работу.

Альтернатива - команда `queue:work` с ключом `--daemon`. В этом случае фреймворк будет загружен один раз и при переходе к следующей задаче не будет перезагружаться, а исполнит её после выполнения предыдущей. В этом случае мы сильно экономим CPU, но такой подход накладывает определенные ограничения - нужно следить за расходом памяти и рестартовать демона после деплоя (заливки на хостинг) нового кода.

Запустить обработчика очереди в демон-режиме:

```
php artisan queue:work connection --daemon

php artisan queue:work connection --daemon --sleep=3

php artisan queue:work connection --daemon --sleep=3 --tries=3
```

Как видно, команда `queue:work` поддерживет те же опции, что и `queue:listen`. Для подробного хелпа по опциям смотрите вывод команды `php artisan help queue:work`.

Деплой приложения с обработчиком-демоном

Самый простой способ корректно деплоить приложение, в которых используются обработчики-демоны - переводить его перед деплоем в режим обслужиания (maintenance mode) командой `php artisan down`. В этом состоянии Laravel не берет новые задачи из очереди, а только продолжает исполнять уже взятые задачи.

Чтобы рестартовать обработчиков очереди, воспользуйтесь командой `queue:restart`

```
php artisan queue:restart
```

Эта команда заставит всех обработчиков-демонов перезапуститься после завершения выполнения их текущей задачи.

> **Примечание:** Команда перезапуска помещается в кэш и каждый обработчик-демон проверяет этот ключ в кэше перед началом выполнения задачи. Если вы используете в качестве кэша APC, то имейте в виду, что по умолчанию он отключен для скриптов, запускаемых из командной строки (в том числе и для обработчиков-демонов), поэтому не забудьте включить его в конфиге APC - `apc.enable_cli=1`

Программирование задач с учетом обработчиков-демонов

При разработке задач, которые будут исполняться обработчиками-демонами, вы должны вручную освобождать все задействованные ресурсы до завершения задачи. Например, если вы обрабатываете изображения при помощи библиотеки GD, освобождайте память при помощи `imagedestroy` в конце задачи.

Также соединение с БД через некоторое время рвется, поэтому не забывайте в начале задачи его возобновлять при помощи `DB::reconnect`.

Push-очереди

Push-очереди дают вам доступ ко всем мощным возможностям, предоставляемым подсистемой очередей Laravel без запуска серверов или фоновых программ. На текущий момент push-очереди поддерживает только драйвер Iron.io. Перед тем, как начать, создайте аккаунт и впишите его данные в `app/config/queue.php`.

Регистрация подписчика push-очереди

После этого вы можете использовать команду `queue:subscribe` Artisan для регистрации URL точки (end-point), которая будет получать добавляемые в очередь задачи.

```
php artisan queue:subscribe queue_name http://foo.com/queue/receive
```

Теперь, когда вы войдёте в ваш профиль Iron, то увидите новую push-очередь и её URL подписки. Вы можете подписать любое число URL на одну очередь. Дальше создайте маршрут для вашей точки `queue/receive` и пусть он возвращает результат вызова метода `Queue::marshal`:

```
Route::post('queue/receive', function()
{
    return Queue::marshal();
});
```

Этот метод позаботится о вызове нужного класса-обработчика задачи. Для помещения задач в push-очередь просто используйте всё тот же метод `Queue::push`, который работает и для обычных очередей.

Незавершённые задачи

Иногда вещи работают не так, как мы хотим, и запланированные задачи, бывает, аварийно завершаются из-за

внутренних ошибок или некорректных входных данных. Даже лучшие из программистов допускают ошибки, это нормально.

Laravel имеет средства для контроля над некорректным завершением задач. Если прошло заданное максимальное количество попыток запуска и задача ни разу не исполнилась до конца, завершившись исключением, она пимещается в базу данных, в таблицу `failed_jobs`. Изменить название таблицы вы можете в конфиге `config/queue.php`.

Данная команда создает миграцию для создания таблицы в вашей базе данных:

```
php artisan queue:failed-table
```

Максимальное число попыток запуска задачи задается параметром `--tries` команды `queue:listen`:

```
php artisan queue:listen connection-name --tries=3
```

Вы можете зарегистрировать слушателя события `Queue::failing`, чтобы, например, получать уведомления по e-mail, что что-то в подсистеме очередей у вас идет не так:

```
Queue::failing(function($connection, $job, $data)
{
    //
});
```

Вы также можете создать в классе команды метод `failed`, позволяющий выполнить какие-либо действия при неудачной попытке выполнения задачи:

```
public function failed()
{
    //
}
```

Получение незавершённых задач

Список всех незавершённых задач с их ID вам покажет команда `queue:failed`:

```
php artisan queue:failed
```

Вы можете вручную рестартовать задачу по её ID:

```
php artisan queue:retry 5
```

Если вы хотите удалить задачу из списка незавершённых, используйте `queue:forget`:

```
php artisan queue:forget 5
```

Чтобы очистить весь список незавершённых задач, используйте `queue:flush`:

```
php artisan queue:flush
```

Настройка

Протокол HTTP не имеет средств для фиксации своего состояния. Сессии - способ сохранения информации (например, ID залогиненного пользователя) между отдельными HTTP-запросами. Laravel поставляется со множеством различных механизмов сессий, доступных через единое API. Изначально существует поддержка таких систем, как Memcached, Redis и СУБД.

Настройки сессии содержатся в файле `config/session.php`. Обязательно просмотрите параметры, доступные вам - они хорошо документированы. По умолчанию Laravel использует драйвер `native`, который подходит для большинства приложений.

Прежде чем использовать Redis, необходимо установить пакет `predis/predis` версии `~1.0` через Composer.

> **Примечание:** Если вы хотите, что бы все данные, хранящиеся в сессиях, были зашифрованы, установите параметр `encrypt` в значение `true` в настройках.

Зарезервированные имена ключей

Laravel использует имя ключа `flash` для внутренних целей, поэтому вы не дожны добавлять в сессию данные с таким ключом.

Использование сессий

Сохранение переменной в сессии

```
Session::put('key', 'value');
```

Добавление элемента к переменной-массиву

```
Session::push('user.teams', 'developers');
```

Чтение переменной сессии

```
$value = Session::get('key');
```

Чтение переменной со значением по умолчанию

```
$value = Session::get('key', 'default');

$value = Session::get('key', function() { return 'дефолтное значение'; });
```

Чтение переменной и удаление её

```
$value = Session::pull('key', 'дефолтное значение');
```

Получение всех переменных сессии

```
$data = Session::all();
```

Проверка существования переменой

```
if (Session::has('users'))
{
    //
}
```

Удаление переменной из сессии

```
Session::forget('key');
```

Удаление всех переменных

```
Session::flush();
```

Присвоение сессии нового идентификатора

```
Session::regenerate();
```

Одноразовые flash-данные

Иногда вам нужно сохранить переменную только для следующего запроса, после выполнения которого она должна быть автоматически удалена. Это нужно, например, для передачи ошибок валидации в форму. Вы можете сделать это методом `Session::flash` (flash англ. - вспышка - прим. пер.):

```
Session::flash('key', 'value');
```

Продление всех одноразовых переменных ещё на один запрос

```
Session::reflash();
```

Продление только отдельных переменных

```
Session::keep(array('username', 'email'));
```

Сессии в базах данных

При использовании драйвера `database` вам нужно создать таблицу, которая будет содержать данные сессий. Ниже пример такого объявления с помощью конструктора таблиц (`Schema`):

```
Schema::create('sessions', function($table)
{
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

Либо вы можете использовать Artisan-команду `session:table` для создания этой миграции:

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

Драйверы

"Драйвер" определяет, где будут храниться данные для каждой сессии. Laravel поставляется с целым набором замечательных драйверов:

- `native` - использует встроенные средства PHP для работы с сессиями.
- `cookie` - данные хранятся в виде зашифрованных cookies.
- `database` - хранение данных в БД, используемой приложением.
- `memcached` и `redis` - используются быстрые кэширующие хранилища пар ключ/значение - memcached или redis.
- `array` - данные содержатся в виде простых массивов PHP и не будут сохраняться между запросами.

  Примечание: драйвер `array` обычно используется для [юнит-тестов], так как он на самом деле не сохраняет данные для последующих запросов.

Шаблоны Blade

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

Создание шаблона Blade

```
<!-- resources/views/layouts/master.blade.php -->

<html>
    <body>
        @section('sidebar')
            Это - главный сайдбар.
        @show

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

Использование шаблона Blade

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

@section('sidebar')
    @parent

    <p>Этот элемент будет добавлен к главному сайдбару.</p>
@stop

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

Заметьте, что шаблоны, которые расширяют другой Blade-шаблон с помощью `extend`, просто перекрывают секции последнего. Старое (перекрытое) содержимое может быть выведено директивой `@@parent`.

Иногда - например, когда вы не уверены, что секция была определена - вам может понадобиться указать значение по умолчанию для директивы `@yield`. Вы можете передать его вторым аргументом:

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

Другие директивы Blade

Вывод переменной

```
Привет, {{ $name }}.

Текущее время эпохи UNIX: {{ time() }}.
```

Вывод переменной с проверкой на существование

Иногда вам нужно вывести переменную, которая может быть определена, а может быть нет. Например:

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

Но вам не обязательно писать тернарный оператор, Blade позволяет записать это короче:

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

Вывод текста с фигурными скобками

Если вам нужно вывести текст с фигурными скобками - управляющими символами Blade - вы можете поставить префикс @:

```
@{{ Этот текст не будет обрабатываться шаблонизатором Blade }}
```

По умолчанию весь выводимый контент экранируется, т.е. все элементы и сущности HTML показываются как есть,

вместо того, чтобы обрабатываться браузером. Если вы не хотите, что бы данные экранировались, используйте следующий синтаксис:

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

> **Примечание:** Будьте осторожны с выводом контента, который получен от пользователей. Всегда используйте двойные фигурные скобки для экранирования возможных элементов и сущностей HTML.

Директива If

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

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

Циклы

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

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

@forelse($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

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

Подшаблоны

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

Вы также можете передать массив переменных во включаемый шаблон:

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

Перезапись секций

Для полной перезаписи секции можно использовать директиву `overwrite`:

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

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

Строки файлов локализации

```
@lang('language.line')

@choice('language.line', 1);
```

Расширение Blade

Blade позволяет создавать свои управляющие структуры. Компилятор Blade располагает двумя хелперами: - `createPlainMatcher` используется для директив, не имеющих аргументов, таких, как `@endif` и `@stop`. - `createMatcher` используется для директив с аргументами.

Вот, например, код, создающий директиву `@datetime($var)` , которая аналогична вызову метода `format()` у переменной `$var`:

```
Blade::extend(function($view, $compiler)
{
    $pattern = $compiler->createMatcher('datetime');

    return preg_replace($pattern, '$1<?php echo $2->format(\'m/d/Y H:i\'); ?>', $view);
});
```

Введение

Laravel построен с учётом того, что современная профессиональная разработка немыслима без юнит-тестирования. Поддержка PHPUnit доступна "из коробки", а файл `phpunit.xml` уже настроен для вашего приложения.

Папка `tests` уже содержит файл теста для примера. После установки нового приложения Laravel просто выполните команду `phpunit` для запуска процесса тестирования.

Написание и запуск тестов

Для создания теста просто создайте новый файл в папке `tests`. Класс теста должен наследовать класс `TestCase`. Вы можете объявлять методы тестов как вы обычно объявляете их для PHPUnit.

Пример тестового класса

```
class FooTest extends TestCase {

    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

Вы можете запустить все тесты в вашем приложении командой `phpunit` в терминале.

> **Примечание:** если вы определили собственный метод `setUp`, не забудьте вызвать `parent::setUp`.

Тестовое окружение

Во время выполнения тестов Laravel автоматически установит текущую среду в `testing`. Кроме этого Laravel подключит настройки тестовой среды для сессии (`session`) и кэширования (`cache`). Оба эти драйвера устанавливаются в `array`, что позволяет данным существовать в памяти, пока работают тесты. Вы можете свободно создать любое другое тестовое окружение по необходимости.

Переменные тестовой среды исполнения (`testing`) можно изменить в файле `phpunit.xml`.

Обращение к URL

Вызов URL из теста

Вы можете легко вызвать любой ваш URL методом `call`:

```
$response = $this->call('GET', 'user/profile');

$response = $this->call($method, $uri, $parameters, $files, $server, $content);
```

После этого вы можете обращаться к свойствам объекта `Illuminate\Http\Response`:

```
$this->assertEquals('Hello World', $response->getContent());
```

Вызов контроллера из теста

Вы также можете вызвать из теста любой контроллер.

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

> **Примечание:** Нет необходимости указывать полный неймспейс в методе `action` - `App\Http\Controllers` можно опустить.

Метод `getContent` вернёт содержимое-строку ответа роута или контроллера. Если был возвращён `View` вы можете получить его через свойство `original`:

```
$view = $response->original;

$this->assertEquals('John', $view['name']);
```

Для вызова HTTPS-маршрута можно использовать метод `callSecure`:

```
$response = $this->callSecure('GET', 'foo/bar');
```

Тестирование фасадов

При тестировании вам может потребоваться отловить вызов (mock a call) к одному из статических классов-фасадов Laravel. К примеру, у вас есть такой контроллер:

```
public function getIndex()
{
    Event::fire('foo', ['name' => 'Дейл']);

    return 'All done!';
}
```

Вы можете отловить обращение к `Event` с помощью метода `shouldReceive` этого фасада, который вернёт объект Mockery.

Мок (mocking) фасада Event

```
public function testGetIndex()
{
    Event::shouldReceive('fire')->once()->with(['name' => 'Дейл']);

    $this->call('GET', '/');
}
```

> **Примечание:** не делайте этого для объекта `Request`. Вместо этого передайте желаемый ввод методу `call` во время выполнения вашего теста.

Проверки (assertions)

Laravel предоставляет несколько `assert`-методов, чтобы сделать ваши тесты немного проще.

Проверка на успешный запрос

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertResponseOk();
}
```

Проверка статуса ответа

```
$this->assertResponseStatus(403);
```

Проверка переадресации в ответе

```
$this->assertRedirectedTo('foo');

$this->assertRedirectedToRoute('route.name');

$this->assertRedirectedToAction('Controller@method');
```

Проверка наличия данных в шаблоне

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertViewHas('name');
    $this->assertViewHas('age', $value);
```

```
}
```

Проверка наличия данных в сессии

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHas('name');
    $this->assertSessionHas('age', $value);
}
```

Проверка на наличие ошибок в сессии

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHasErrors();

    // Asserting the session has errors for a given key...
    $this->assertSessionHasErrors('name');

    // Asserting the session has errors for several keys...
    $this->assertSessionHasErrors(array('name', 'age'));
}
```

Проверка на наличие "старого пользовательского ввода"

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertHasOldInput();
}
```

Вспомогательные методы

Класс `TestCase` содержит несколько вспомогательных методов для упрощения тестирования вашего приложения.

Установка текущего авторизованного пользователя

Вы можете установить текущего авторизованного пользователя с помощью метода be:

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

$this->be($user);
```

Вы можете заполнить вашу БД начальными данными изнутри теста методом `seed`.

**Заполнение БД тестовыми данными

```
$this->seed();

$this->seed($connection);
```

Больше информации на тему начальных данных доступно в разделе Миграции и начальные данные.

Ресет IoC-контейнера Laravel

Как вы уже знаете, в любой части теста вы можете получить доступ к IoC-контейнеру приложения Laravel при помощи `$this->app`. Объект приложения Laravel обновляется для каждого тестового класса, но не метода. Если вы хотите вручную ресетить объект приложения Laravel в произвольном месте, вы можете это сделать при помощи метода `refreshApplication`. Это сбросит все моки (mock) и другие дополнительные биндинги, которые были сделаны с момента запуска тест-сессии.

Использование валидации

Laravel поставляется с простой, удобной системой валидации (проверки входных данных на соответствие правилам) и получения сообщений об ошибках - классом `Validation`.

Простейший пример валидации

```
$validator = Validator::make(
    array('name' => 'Дейл'),
    array('name' => 'required|min:5')
);
```

Первый параметр, передаваемый методу `make` - данные для проверки. Второй параметр - правила, которые к ним должны быть применены.

Использование массивов для указания правил

Несколько правил могут быть разделены либо прямой чертой (|), либо быть отдельными элементами массива.

```
$validator = Validator::make(
    array('name' => 'Дейл'),
    array('name' => array('required', 'min:5'))
);
```

Проверка нескольких полей

```
$validator = Validator::make(
    array(
        'name' => 'Дейл',
        'password' => 'плохойпароль',
        'email' => 'email@example.com'
    ),
    array(
        'name' => 'required',
        'password' => 'required|min:8',
        'email' => 'required|email|unique'
    )
);
```

Как только был создан экземпляр `Validator`, метод `fails` (или `passes`) может быть использован для проведения проверки.

```
if ($validator->fails())
{
    // Переданные данные не прошли проверку
}
```

Если `Validator` нашёл ошибки, вы можете получить его сообщения таким образом:

```
$messages = $validator->messages();
```

Вы также можете получить массив правил, данные которые не прошли проверку, без самих сообщений:

```
$failed = $validator->failed();
```

Проверка файлов

Класс `Validator` содержит несколько изначальных правил для проверки файлов, такие как `size`, `mimes` и другие. Для выполнения проверки над файлами просто передайте эти файлы вместе с другими данными.

Хук после валидации

Laravel после завершения валидации может запустить вашу функцию-замыкание, в которой вы можете, например,

проверить что-то особенное или добавить какое-то своё сообщение об ошибке. Для этого служит метод `after()`:

```
$validator = Validator::make(...);

$validator->after(function($validator)
{
    if ($this->somethingElseIsInvalid())
    {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails())
{
    //
}
```

Вы можете добавить несколько `after`, если это нужно.

Валидация в контроллерах

Писать полный код валидации каждый раз, когда нужно провалидировать данные - это неудобно. Поэтому Laravel предоставляет несколько решений для упрощения этой процедуры.

Базовый контроллер `App\Http\Controllers\Controller` включает в себя трейт `ValidatesRequests`, который уже содержит методы для валидации:

```
/**
 * Сохранить пост в блоге.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    //
}
```

Если валидация проходит, код продолжает выполняться. Если нет - бросается исключение `Illuminate\Contracts\Validation\ValidationException`. Если вы не поймаете это исключение, его поймает фреймворк, заполнит flash-переменные сообщениями об ошибках валидации и средиректит пользователя на предыдущую страницу с формой - сам !

В случае AJAX-запроса редиректа не происходит, фреймворк отдает ответ с HTTP-кодом 422 и JSON с ошибками валидации.

Код, приведенный выше, аналогичен вот этому::

```
/**
 * Сохранить пост в блоге.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $v = Validator::make($request->all(), [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    if ($v->fails())
    {
        return redirect()->back()->withErrors($v->errors());
    }

    //
}
```

Изменения формата ошибок

Если вы хотите кастомизировать сообщения об ошибках валидации, которые сохраняются во флэш-переменных сессии при редиректе, перекройте метод `formatValidationErrors` в вашем контроллере:

```
/**
 * {@inheritdoc}
 */
protected function formatValidationErrors(\Illuminate\Validation\Validator $validator)
{
    return $validator->errors()->all();
}
```

Валидация запросов

Для реализации более сложных сценариев валидации вам могут быть удобны так называемые Form Requests. Это специальные классы HTTP-запроса, содержащие в себе логику валидации. Они обрабатывают запрос до того, как он поступит в контроллер.

Чтобы создать класс запроса, используйте artisan-команду `make:request`:

```
php artisan make:request StoreBlogPostRequest
```

Класс будет создан в папке `app/Http/Requests`. Добавьте необходимые правила валидации в его метод `rules`:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ];
}
```

Для того, чтобы фрейморк перехватил запрос перед контроллером, добавьте этот класс в аргументы необходимого метода контроллера:

```
/**
 * Сохранить пост в блоге.
 *
 * @param  StoreBlogPostRequest  $request
 * @return Response
 */
public function store(StoreBlogPostRequest $request)
{
    // Валидация успешно пройдена
}
```

При грамотном использовании валидации запросов вы можете быть уверены, что в ваших контроллерах всегда находятся только отвалидированные входные данные !

В случае неудачной валидации фреймворк заполняет флэш-переменные ошибками валидации и возвращает редирект на предыдущую страницу. В случае AJAX-запроса отдается ответ с кодом 422 и JSON с ошибками валидации.

Контроль доступа

Классы Form Request также содержать метод `authorize`. В этом меторже вы можете проверять, разрешено ли пользователю совершать это действие, обновлять данный ресурс. Например, если пользователь пытается отредактировать комментарий к посту, является ли он его автором ?

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $commentId = $this->route('comment');
```

```
    return Comment::where('id', $commentId)
                ->where('user_id', Auth::id())->exists();
}
```

Обратите внимание на вызов метода `route()` выше. Этод метод дает вам доступ к параметрам в урле (в данном случае это `{comment}`), определенным в роуте:

```
Route::post('comment/{comment}');
```

Если метод `authorize` возвращает `false`, фреймворк формирует ответ с HTTP-кодом 403 и сразу же отсылает его. Метод контроллера не выполняется.

Если ваша логика авторизации и проверки доступа находится в контроллере, просто верните `true` в этом методе:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

Изменения формата ошибок во флэш-переменных

Если вы хотите кастомизировать сообщения об ошибках валидации, которые сохраняются во флэш-переменных сессии при редиректе, перекройте метод `formatValidationErrors` в базовом классе запросов (`App\Http\Requests\Request`):

```
/**
 * {@inheritdoc}
 */
protected function formatValidationErrors(\Illuminate\Validation\Validator $validator)
{
    return $validator->errors()->all();
}
```

Работа с сообщениями об ошибках

После вызова метода `messages` объекта `Validator` вы получите объект `MessageBag`, который имеет набор полезных методов для доступа к сообщеням об ошибках.

Получение первого сообщения для поля

```
echo $messages->first('email');
```

Получение всех сообщений для одного поля

```
foreach ($messages->get('email') as $message)
{
    //
}
```

Получение всех сообщений для всех полей

```
foreach ($messages->all() as $message)
{
    //
}
```

Проверка на наличие сообщения для поля

```
if ($messages->has('email'))
{
    //
}
```

Получение ошибки в заданном формате

```
echo $messages->first('email', '<p>:message</p>');
```

Примечание: по умолчанию сообщения форматируются в вид, который подходит для Twitter Bootstrap.

Получение всех сообщений в заданном формате

```
foreach ($messages->all('<li>:message</li>') as $message)
{
    //
}
```

Ошибки и шаблоны

Как только вы провели проверку, вам понадобится простой способ, чтобы передать ошибки в шаблон. Laravel позволяет удобно сделать это. Например, у нас есть такие роуты:

```
Route::get('register', function()
{
    return View::make('user.register');
});

Route::post('register', function()
{
    $rules = array(...);

    $validator = Validator::make(Input::all(), $rules);

    if ($validator->fails())
    {
        return redirect('register')->withErrors($validator);
    }
});
```

Заметьте, что когда проверки не пройдены, мы передаём объект `Validator` объекту переадресации `Redirect` с помощью метода `withErrors`. Этот метод сохранит сообщения об ошибках в одноразовых flash-переменных сессии, таким образом делая их доступными для следующего запроса.

Однако, заметьте, мы не передаем в `View::make('user.register');` переменные `$errors` в шаблон. Laravel сам проверяет данные сессии на наличие переменных и автоматически передает их шаблону, если они доступны. **Таким образом, важно помнить, что переменная `$errors` будет доступна для всех ваших шаблонов всегда, при любом запросе.**. Это позволяет вам считать, что переменная `$errors` всегда определена и может безопасно использоваться. Переменная `$errors` - экземпляр класса `MessageBag`.

Таким образом, после переадресации вы можете прибегнуть к автоматически установленной в шаблоне переменной `$errors`:

```
<?php echo $errors->first('email'); ?>
```

Именованные MessageBag

Если у вас есть несколько форм на странице, то вы можете выбрать имя объекта MessageBag, в котором будут возвращаться тексты ошибок, чтобы вы могли их корректно отобразить для нужной формы.

```
return redirect('register')->withErrors($validator, 'login');
```

Получить текст ошибки из MessageBag с именем login:

```
<?php echo $errors->login->first('email'); ?>
```

Правила проверки

Ниже список всех доступных правил и их функции:

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Before (Date)
- Between
- Boolean

- Confirmed
- Date
- Date Format
- Different
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer
- IP Address
- Max
- MIME Types
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required With All
- Required Without
- Required Without All
- Same
- Size
- Timezone
- Unique (Database)
- URL

accepted

Поле должно быть в значении *yes*, *on* или *1*. Это полезно для проверки принятия правил и лицензий.

active_url

Поле должно быть корректным URL, доступным через функцию checkdnsrr.

after:*date*

Поле должно быть датой, более поздней, чем *date*. Строки приводятся к датам функцией `strtotime`.

alpha

Поле можно содержать только алфавитные символы.

alpha_dash

Поле можно содержать только алфавитные символы, цифры, знаки подчёркивания (_) и дефисы (-).

alpha_num

Поле можно содержать только алфавитные символы и цифры.

array

Поле должно быть массивом.

before:*date*

Поле должно быть датой, более ранней, чем *date*. Строки приводятся к датам функцией `strtotime`.

between:*min*,*max*

Поле должно быть числом в диапазоне от *min* до *max*. Строки, числа и файлы трактуются аналогично правилу `size`.

boolean

Поле должно быть логическим (булевым). Разрешенные значения: `true`, `false`, `1`, `0`, `"1"` и `"0"`.

confirmed

Значение поля должно соответствовать значению поля с этим именем, плюс `foo_confirmation`. Например, если

проверяется поле password, то на вход должно быть передано совпадающее по значению поле password_confirmation.

date

Поле должно быть правильной датой в соответствии с функцией strtotime.

date*format:*format_

Поле должно подходить под формату даты *format* в соответствии с функцией date_parse_from_format.

different:*field*

Значение проверяемого поля должно отличаться от значения поля *field*.

email

Поле должно быть корректным адресом e-mail.

exists:*table*,*column*

Поле должно существовать в заданной таблице базе данных.

Простое использование:

```
'state' => 'exists:states'
```

Указание имени поля в таблице:

```
'state' => 'exists:states,abbreviation'
```

Вы также можете указать больше условий, которые будут добавлены к запросу "WHERE":

```
'email' => 'exists:staff,email,account_id,1'
```

image

Загруженный файл должен быть изображением в формате jpeg, png, bmp, gif или svg.

in:*foo*,*bar*,...

Значение поля должно быть одним из перечисленных (*foo*, *bar* и т.д.).

integer

Поле должно иметь корректное целочисленное значение.

ip

Поле должно быть корректным IP-адресом.

max:*value*

Значение поля должно быть меньше или равно *value*. Строки, числа и файлы трактуются аналогично правилу size.

mimes:*foo*,*bar*,...

MIME-тип загруженного файла должен быть одним из перечисленных.

Простое использование:

```
'photo' => 'mimes:jpeg,bmp,png'
```

min:*value*

Значение поля должно быть более *value*. Строки, числа и файлы трактуются аналогично правилу size.

not*in:*foo,bar_,...

Значение поля **не** должно быть одним из перечисленных (*foo*, *bar* и т.д.).

numeric

Поле должно иметь корректное числовое или дробное значение.

regex:*pattern*

Поле должно соответствовать заданному регулярному выражению.

Внимание: при использовании этого правила может быть нужно перечислять другие правила в виде элементов массива, особенно если выражение содержит символ вертикальной черты (|).

required

Проверяемое поле должно иметь непустое значение.

required*if:*field,value_,...

Проверяемое поле должно иметь непустое значение, если другое поле *field* имеет любое из значений *value*.

required*with:*foo,bar_,...

Проверяемое поле должно иметь непустое значение, но только если присутствует хотя бы одно из перечисленных полей (*foo*, *bar* и т.д.).

required*with*all:*foo*,*bar*,...

Проверяемое поле должно иметь непустое значение, но только если присутствуют все перечисленные поля (*foo*, *bar* и т.д.).

required*without:*foo,bar_,...

Проверяемое поле должно иметь непустое значение, но только если **не** присутствует хотя бы одно из перечисленных полей (*foo*, *bar* и т.д.).

required*without*all:*foo*,*bar*,...

Проверяемое поле должно иметь непустое значение, но только если **не** присутствуют все перечисленные поля (*foo*, *bar* и т.д.).

same:*field*

Поле должно иметь то же значение, что и поле *field*.

size:*value*

Поле должно иметь совпадающий с *value* размер. **Для строк** это обозначает длину, **для чисел** - число, **для файлов** - размер в килобайтах.

timezone

Поле должно содержать идентификатор часового пояса (таймзоны), один из перечисленных в php-функции `timezone_identifiers_list`

unique:*table*,*column*,*except*,*idColumn*

Значение поля должно быть уникальным в заданной таблице базы данных. Если `column` не указано, то будет использовано имя поля.

Простое использование

```
'email' => 'unique:users'
```

Указание имени поля в таблице

```
'email' => 'unique:users,email_address'
```

Игнорирование определённого ID

```
'email' => 'unique:users,email_address,10'
```

Добавление дополнительных условий

Вы также можете указать больше условий, которые будут добавлены к запросу "WHERE"":

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

В правиле выше только строки с account_id равном 1 будут включены в проверку.

url

Поле должно быть корректным URL.

> **Примечание:** используется PHP-функция filter_var

Условные правила

Иногда вам нужно валидировать некое поле **только** тогда, когда оно присутствует во входных данных. Для этого добавьте правило sometimes:

```
$v = Validator::make($data, array(
    'email' => 'sometimes|required|email',
));
```

В примере выше поле для поля email будет запущена валидация только когда $data['email'] существует.

Сложные условные правила

Иногда вам может нужно, чтобы поле имело какое-либо значение только если другое поле имеет значеие, скажем, больше 100. Или вы можете требовать наличия двух полей только, когда также указано третье. Это легко достигается условными правилами. Сперва создайте объект Validator с набором статичных правил, которые никогда не изменяются:

```
$v = Validator::make($data, array(
    'email' => 'required|email',
    'games' => 'required|numeric',
));
```

Теперь предположим, что ваше приложения написано для коллекционеров игр. Если регистрируется коллекционер с более, чем 100 играми, то мы хотим их спросить, зачем им такое количество. Например, у них может быть магазин или может им просто нравится их собирать. Итак, для добавления такого условного правила мы используем метод Validator.

```
$v->sometimes('reason', 'required|max:500', function($input)
{
    return $input->games >= 100;
});
```

Первый параметр этого метода - имя поля, которое мы проверяем. Второй параметр - правило, которое мы хотим добавить, если переданная функция-замыкание (третий параметр) вернёт true. Этот метод позволяет легко создавать сложные правила проверки ввода. Вы можете даже добавлять одни и те же условные правила для нескольких полей одновременно:

```
$v->sometimes(array('reason', 'cost'), 'required', function($input)
{
    return $input->games >= 100;
});
```

> **Примечание:** Параметр $input, передаваемый замыканию - объект Illuminate\Support\Fluent и может использоваться для чтения проверяемого ввода и файлов.

Собственные сообщения об ошибках

Вы можете передать собственные сообщения об ошибках вместо используемых по умолчанию. Если несколько способов это сделать.

Передача своих сообщений в Validator

```
$messages = array(
    'required' => 'Поле :attribute должно быть заполнено.',
);

$validator = Validator::make($input, $rules, $messages);
```

> **Примечание:** строка `:attribute` будет заменена на имя проверяемого поля. Вы также можете использовать и другие строки-переменные.

Использование других переменных-строк

```
$messages = array(
    'same'    => 'Значения :attribute и :other должны совпадать.',
    'size'    => 'Поле :attribute должно быть ровно exactly :size.',
    'between' => 'Значение :attribute должно быть от :min и до :max.',
    'in'      => 'Поле :attribute должно иметь одно из следующих значений: :values',
);
```

Указание собственного сообщения для отдельного поля

Иногда вам может потребоваться указать своё сообщение для отдельного поля.

```
$messages = array(
    'email.required' => 'Нам нужно знать ваш e-mail адрес!',
);
```

Указание собственных сообщений в файле локализации

Также можно определять сообщения валидации в файле локализации вместо того, чтобы передавать их в `Validator` напрямую. Для этого добавьте сообщения в массив `custom` файла локализации `app/lang/xx/validation.php`.

```
'custom' => array(
    'email' => array(
        'required' => 'Нам нужно знать ваш e-mail адрес!',
    ),
),
```

Собственные правила проверки

Регистрация собственного правила валидации

Laravel изначально содержит множество полезных правил, однако вам может понадобиться создать собственные. Одним из способов зарегистрировать произвольное правило - через метод `Validator::extend`.

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

> **Примечание:** имя правила должно быть в формате_с_подчёркиваниями.

Переданная функция-замыкание получает три параметра: имя проверяемого поля `$attribute`, значение поля `$value` и массив параметров `$parameters` переданных правилу.

Вместо функции в метод `extend` можно передать ссылку на метод класса:

```
Validator::extend('foo', 'FooValidator@validate');
```

Обратите внимание, что вам также понадобится определить сообщение об ошибке для нового правила. Вы можете сделать это либо передавая его в виде массива строк в `Validator`, либо вписав в файл локализации.

Расширение класса `Validator`

Вместо использования функций-замыканий для расширения набора доступных правил вы можете расширить сам класс `Validator`. Для этого создайте класс, который наследует `Illuminate\Validation\Validator`. Вы можете добавить новые методы проверок, начав их имя с `validate`.

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'значение';
    }

}
```

Регистрация нового класса `Validator`

Затем вам нужно зарегистрировать это расширение валидации. Сделать это можно, например, в вашем сервис-провайдере или в ваших старт-файлах.

```
Validator::resolver(function($translator, $data, $rules, $messages)
{
    return new CustomValidator($translator, $data, $rules, $messages);
});
```

Иногда при создании своего класса валидации вам может понадобиться определить собственные строки-переменные (типа ":foo") для замены в сообщениях об ошибках. Это делается путём создания класса, как было описано выше, и добавлением функций с именами вида `replaceXXX`.

```
protected function replaceFoo($message, $attribute, $rule, $parameters)
{
    return str_replace(':foo', $parameters[0], $message);
}
```

Если вы хотите добавить свое сообщение без использования `Validator::extend`, вы можете использовать метод `Validator::replacer`:

```
Validator::replacer('rule', function($message, $attribute, $rule, $parameters)
{
    //
});
```

Настройка

Laravel делает процесс соединенеия с БД и выполнение запросов очень простым. Настройки работы с БД хранятся в файле `config/database.php`. Здесь вы можете указать все используемые вами соединения к БД, а также указать, 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, Postgres, SQLite и SQL Server.

Раздельное чтение и запись

Возможно, иногда вам понадобится использовать одно соединения для выполнения запроса SELECT, а другое для запросов INSERT, UPDATE и DELETE. Laravel максимально упрощает этот процесс, причем не важно, что вы используете - сырые запросы, например `DB::select()`, Query Builder или Eloquent ORM.

Рассмотрим пример конфигурации:

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
],
```

Заметьте, два ключа были добавлены в массив настроек: `read` и `write`. Оба из них содержат единственный ключ: `host`. Остальные настройки для обеих этих операций одинаковы и берутся из массива `mysql`. При необходимости изменения и других настроек (не только `host`), просто добавьте и другие ключи в `read` и `write`, которые перезапишут такие же настройки в главном массиве. Таким образом, `192.168.1.1` будет использоваться для чтения, а `192.168.1.2` - для записи. Остальные настройки совпадают.

Выполнение запросов

Как только вы настроили соединение с базой данных, вы можете выполнять запросы используя фасад `DB`.

Выполнение запроса SELECT

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

Метод `select` всегда возвращает массив результатов.

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');
```

> **Примечание:** запросы `update` и `delete` возвращают число затронутых строк.

Выполнение запроса другого типа

```
DB::statement('drop table users');
```

Реагирование на выполнение запросов

Вы можете добавить собственный обработчик, вызываемый при выполнении очередного запроса, с помощью метода `DB::listen`:

```
DB::listen(function($sql, $bindings, $time)
{
    //
});
```

Транзакции

Для выполнения запросов внутри одной транзакции, воспользуйтесь методом `transaction`:

```
DB::transaction(function()
{
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

> **Примечание:** Любая ошибка, полученная при выполнении запросов транзакции, отменит все изменения, вызванные ей.

Иногда вам понадобится начать транзакцию вручную:

```
DB::beginTransaction();
```

Отмена транзакции и изменений, вызванных её выполнением:

```
DB::rollback();
```

Завершение и подтверждение транзакцию:

```
DB::commit();
```

Доступ к соединениям

При использовании нескольких подключений к БД, вы можете получить к ним достум через метод `DB::connection`:

```
$users = DB::connection('foo')->select(...);
```

Вы также можете получить низкоуровневый объект PDO этого подключения:

```
$pdo = DB::connection()->getPdo();
```

Иногда вам может понадобиться переподключиться к БД и вы можете сделать это так:

```
DB::reconnect('foo');
```

Если вам нужно отключиться от БД - например, чтобы не превысить лимит `max_connections` в БД, вы можете воспользоваться методом `disconnect`:

```
DB::disconnect('foo');
```

Журнал запросов

По умолчанию, Laravel записывает все SQL-запросы в памяти, выполненные в рамках текущего HTTP-запроса. Однако, в некоторых случаях, как например при вставке большого количества записей, это может быть слишком ресурсозатратно. Для отключения журнала вы можете использовать метод `disableQueryLog`:

```
DB::connection()->disableQueryLog();
```

Для получения массива выполненных запросов используйте метод `getQueryLog`:

```
   $queries = DB::getQueryLog();
```

Введение

Query Builder - конструктор запросов - предоставляет удобный, выразительный интерфейс для создания и выполнения запросов к базе данных. Он может использоваться для выполнения большинства типов операций и работает со всеми подерживаемыми СУБД.

> **Примечание:** конструктор запросов Laravel использует средства PDO для защиты вашего приложения от SQL-инъекций. Нет необходимости экранировать строки перед их передачей в запрос.

Выборка (SELECT)

Получение всех записей таблицы

```
$users = DB::table('users')->get();

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

Ограничение результатов выборки

Вы можете отобрать некоторое количество результатов из выборки:

```
DB::table('users')->chunk(100, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

Вы можете остановить отбор результатов в чанк, вернув `false` из функции-замыкания.

```
DB::table('users')->chunk(100, function($users)
{
    //

    return false;
});
```

Получение одной записи

```
$user = DB::table('users')->where('name', 'Джон')->first();

var_dump($user->name);
```

Получение одного поля из записей

```
$name = DB::table('users')->where('name', 'Джон')->pluck('name');
```

Получение списка всех значений одного поля

```
$roles = DB::table('roles')->lists('title');
```

Этот метод вернёт массив всех заголовков (title). Вы можете указать произвольный ключ для возвращаемого массива:

```
$roles = DB::table('roles')->lists('title', 'name');
```

Указание полей для выборки

```
$users = DB::table('users')->select('name', 'email')->get();

$users = DB::table('users')->distinct()->get();

$users = DB::table('users')->select('name as user_name')->get();
```

Добавление полей к созданному запросу

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Использование фильтрации WHERE

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

Условия ИЛИ:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'Джон')
                    ->get();
```

Фильтрация по интервалу значений

```
$users = DB::table('users')
                    ->whereBetween('votes', array(1, 100))->get();
```

Фильтрация по совпадению с массивом значений

```
$users = DB::table('users')
                    ->whereIn('id', array(1, 2, 3))->get();

$users = DB::table('users')
                    ->whereNotIn('id', array(1, 2, 3))->get();
```

Поиск неустановленных значений (NULL)

```
$users = DB::table('users')
                    ->whereNull('updated_at')->get();
```

Использование By, Group By и Having

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

Смещение от начала и лимит числа возвращаемых строк

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Объединения (JOIN)

Конструктор запросов может быть использован для выборки данных из нескольких таблиц через JOIN. Посмотрите на примеры ниже.

Простое объединение

```
DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.id', 'contacts.phone', 'orders.price');
```

Объединение типа LEFT JOIN

```
DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
        ->get();
```

Вы можете указать более сложные условия:

```
DB::table('users')
        ->join('contacts', function($join)
        {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

Внутри `join()` можно использовать `where` и `orWhere`:

```
DB::table('users')
        ->join('contacts', function($join)
        {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Сложные выражения WHERE

Иногда вам нужно сделать выборку по более сложным параметрам, таким как "существует ли" или вложенная группировка условий. Конструктор запросов Laravel справится и с такими запросами.

Группировка условий

```
DB::table('users')
            ->where('name', '=', 'Джон')
            ->orWhere(function($query)
            {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Админ');
            })
            ->get();
```

Команда выше выполнит такой SQL:

```
select * from users where name = 'Джон' or (votes > 100 and title <> 'Админ')
```

Проверка на существование

```
DB::table('users')
            ->whereExists(function($query)
            {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

Эта команда выше выполнит такой запрос:

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

Аггрегатные функции

Конструктор запросов содержит множество аггрегатных методов, таких как `count`, `max`, `min`, `avg` и `sum`.

Использование аггрегатных функций

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');

$price = DB::table('orders')->min('price');

$price = DB::table('orders')->avg('price');

$total = DB::table('users')->sum('votes');
```

Сырые SQL-выражения

Иногда вам может быть нужно использовать уже готовое SQL-выражение в вашем запросе. Такие выражения вставляются в запрос напрямую в виде строк, поэтому будьте внимательны и не создавайте возможных точек для SQL-инъекций. Для создания сырого выражения используется метод `DB::raw`.

Использование SQL-выражения в конструкторе запросов

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

Вставка (INSERT)

Вставка записи в таблицу

```
DB::table('users')->insert(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

Вставка записи и получение её нового ID

Если таблица имеет автоинкрементный индекс, то можно использовать метод `insertGetId` для вставки записи и получения её порядкового номера.

```
$id = DB::table('users')->insertGetId(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

> **Примечание:** при использовании PostgreSQL автоинкрементное поле должно иметь имя "id".

Вставка нескольких записей одновременно

```
DB::table('users')->insert(array(
    array('email' => 'taylor@example.com', 'votes' => 0),
    array('email' => 'dayle@example.com', 'votes' => 0),
));
```

Обновление (UPDATE)

Обновление записей в таблице

```
DB::table('users')
            ->where('id', 1)
            ->update(array('votes' => 1));
```

Увеличение или уменьшение значения поля

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

Вы также можете указать дополнительные поля для изменения:

```
DB::table('users')->increment('votes', 1, array('name' => 'Джон'));
```

Удаление (DELETE)

Удаление записей из таблицы

```
DB::table('users')->where('votes', '<', 100)->delete();
```

Удаление всех записей

```
DB::table('users')->delete();
```

Очистка таблицы

```
DB::table('users')->truncate();
```

> Очистка таблицы аналогична удалению всех её записей, а также сбросом счётчика автоинкремент-поля - прим. пер.

Слияние (UNION)

Конструктор запросов позволяет создавать слияния двух запросов вместе.

```
$first = DB::table('users')->whereNull('first_name');

$users = DB::table('users')->whereNull('last_name')->union($first)->get();
```

Также существует метод `unionAll` с аналогичными параметрами.

Блокирование (lock) данных

SELECT с 'shared lock':

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

SELECT с 'lock for update':

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

Кэширование запросов

Вы можете легко закэшировать запрос методом `remember`:

Кэширование запросов

```
$users = DB::table('users')->remember(10)->get();
```

В этом примере результаты выборки будут сохранены в кэше на 10 минут. В течении этого времени данный запрос не будет отправляться к СУБД - вместо этого результат будет получен из системы кэширования, указанного по умолчанию в вашем файле настроек.

Если система кэширования, которую вы выбрали, поддерживает тэги, вы можете их указать в запросе:

```
$users = DB::table('users')->cacheTags(array('people', 'authors'))->remember(10)->get();
```

Введение

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

Прежде чем начать настройте ваше соединение с БД в файле `config/database.php`.

Использование ORM

Для начала создадим модель Eloquent. Модели обычно располагаются в папке app, но вы можете поместить в любое место, в котором работает автозагрузчик в соответствии с вашим файлом `composer.json`. Модели Eloquent должны расширять класс `Illuminate\Database\Eloquent\Model`.

Создание модели Eloquent

```
class User extends Model {}
```

Так же модель Eloquent можно создать Artisan-командой `make:model`:

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

Заметьте, что мы не указали, какую таблицу Eloquent должен привязать к нашей модели. Если это имя не указано явно, то будет использовано имя класса в нижнем регистре и во множественном числе. В нашем случае Eloquent предположит, что модель `User` хранит свои данные в таблице `users`. Вы можете указать произвольную таблицу, определив свойство `table` в классе модели:

```
class User extends Model {

    protected $table = 'my_users';

}
```

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

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

Получение всех моделей (записей)

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

Получение записи по первичному ключу

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

var_dump($user->name);
```

> **Примечание:** Все методы, доступные в конструкторе запросов, также доступны при работе с моделями Eloquent.

Получение модели по первичному ключу с возбуждением исключения

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

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

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

Для регистрации обработчика ошибки подпишитесь на событие `ModelNotFoundException`:

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
});
```

Построение запросов в моделях Eloquent

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

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

```
}
```

Аггрегатные функции в Eloquent

Конечно, вам также доступны аггрегатные функции.

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

Если у вас не получается создать нужный запрос с помощью методов конструктора, то можно использовать метод `whereRaw`:

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

Обработка результата по частям

Если в вашей eloquent-выборке получилось очень много элементов, то обрабатывая их в цикле foreach вы можете израсходовать всю доступную оперативную память. Чтобы этого не произошло, используйте метод chunk таким образом:

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

Данный код вынимает данные порциями по 200 (первый аргумент) записей. Обработка записи производится в функции-замыкании, которая передается вторым аргументом.

Указание имени соединения с БД

Иногда вам нужно указать, какое подключение должно быть использовано при выполнении запроса Eloquent - просто используйте метод on:

```
$user = User::on('имя-соединения')->find(1);
```

Если вы используете отдельные соединения на чтение и запись, вы можете принудительно прочитать данные по соединению, которое осуществляет запись:

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

Массовое заполнение

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

Для начала определите в классе модели свойство `fillable` или `guarded`.

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

Указание доступных к заполнению атрибутов

```
class User extends Model {

    protected $fillable = ['first_name', 'last_name', 'email'];

}
```

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

Противоположность `fillable` - свойство `guarded`, которое содержит список запрещённых к заполнению полей.

Указание охраняемых (guarded) атрибутов модели

```
class User extends Model {

    protected $guarded = ['id', 'password'];

}
```

> **Примечание:** Даже если вы используете `guarded`, по возможности избегайте массового присваивания `Input::get()` модели.

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 специальным значением.

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

Вставка, обновление, удаление

Для создания новой записи в БД просто создайте экземпляр модели и вызовите метод `save`.

Сохранение новой модели

```
$user = new User;

$user->name = 'Джон';

$user->save();
```

> **Примечание:** обычно ваши модели Eloquent содержат автоматически увеличивающиеся (autoincrementing) числовые ключи. Однако если вы хотите использовать собственные ключи, установите свойство `incrementing` класса модели в значение `false`.

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

Установка охранных свойств модели

```
class User extends Model {

    protected $guarded = ['id', 'account_id'];

}
```

Создание модели

```
$user = User::create(['name' => 'Джон']);

// Создать нового пользователя в базе данных
$user = User::create(['name' => 'John']);

// Получить пользователя с данным именем, а если он отсутствует - создать его в базе данных
$user = User::firstOrCreate(['name' => 'John']);

// Получить пользователя с данным именем, а если он отсутствует - создать его объект (для сохранения в БД нужно будет сделать `$user->save()`)
$user = User::firstOrNew(['name' => 'John']);
```

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 используйте метод push.

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

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

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

> **Примечание:** В случае подобного апдейта через конструктор запросов, обработчики событий модели запускаться не будут.

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, вы также можете выполнять удаление на наборе моделей:

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

Обновление времени изменения модели

Если вам нужно просто обновить время изменения записи - используйте метод touch:

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

Псевдоудаление (soft deleting)

Когда вы удаляете модель с включенным softDelete, она на самом деле остаётся в базе данных, однако в её поле deleted_at записывается текущее время. Для включения псевдоудалений добавьте в модель трейт SoftDeletes:

use Illuminate\Database\Eloquent\SoftDeletes;

```
class User extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

}
```

Для добавления поля deleted_at к таблице можно в миграции использовать метод softDeletes:

```
$table->softDeletes();
```

Теперь когда вы вызовите метод delete, поле deleted_at будет установлено в значение текущего времени. При запросе моделей, использующих псевдоудаление, «удалённые» модели не будут включены в результат запроса.

Включение удалённых моделей в результат выборки

Для отображения всех моделей, в том числе удалённых, используйте метод withTrashed.

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

Он также работает в отношениях модели:

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

Если вы хотите получить **только** удалённые модели, вызовите метод onlyTrashed:

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

Для восстановления псевдоудалённой модели в активное состояние используется метод `restore`:

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

Вы также можете использовать его в запросе:

```
User::withTrashed()->where('account_id', 1)->restore();
```

Метод `restore` можно использовать и в отношениях:

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

Если вы хотите полностью удалить модель из БД, используйте 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())
{
    //
}
```

Дата и время

По умолчанию Eloquent автоматически поддерживает поля `created_at` и `updated_at`, записывая в них, соответственно, дату и время (timestamp) создания и обновления строки в базе данных. Просто добавьте эти timestamp-поля к таблице и Eloquent позаботится об остальном. Если вы не хотите, чтобы он поддерживал их, добавьте свойство `timestamps` равное `false` к классу модели.

Отключение автоматических полей времени

```
class User extends Model {

    protected $table = 'users';

    public $timestamps = false;

}
```

Для настройки форматов времени перекройте метод `getDateFormat`:

Использование собственного формата времени

```
class User extends Model {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

Заготовки запросов (query scopes)

Заготовки (скоупы) позволяют вам повторно использовать логику запросов в моделях. Для создания скоупа просто начните имя метода со scope:

Создание заготовки запроса

```
class User extends Model {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Использование скоупа

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

Динамические скоупы

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

```
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();
```

Глобальные заготовки запросов

Иногда вам нужно определить скоуп, который должен быть доступен во всех моделях. Например, псевдоудаление (soft delete) в Laravel построено на этой фиче. Глобальные заготовки запросов создаются путем комбинации PHP-трейтов и имплементации `Illuminate\Database\Eloquent\ScopeInterface`.

Для начала, определим трейт. Для примера, рассмотрим реализацию псевдоудаления (soft delete):

```
trait SoftDeletes {

    /**
     * Загрузка трейта soft delete.
     *
     * @return void
     */
    public static function bootSoftDeletes()
    {
        static::addGlobalScope(new SoftDeletingScope);
    }

}
```

Если в Eloquent-модель подключается трейт, то при загрузке модели вызывается метод трейта, который называется по принципу `bootNameOfTrait` (т.е. в случае трейта `SoftDeletes` - `bootSoftDeletes()`). Этот метод - подходящее место для регистрации глобальной заготовоки запроса. Этот скоуп должен быть реализацией `ScopeInterface` и иметь два метода - `apply` и `remove`.

Метод `apply` принимает объект конструктора запросов `Illuminate\Database\Eloquent\Builder` и `Model`, к которой надо применить скоуп и добавляет в него все необходимые запросы `where`. Метод `remove` тоже принимает объект `Builder` и `Model` и отвечает за действия, обратные тем, которые мы сделали в `apply`. Взгляните на пример:

```
/**
 * Берем только те записи, поле `deleted_at` для которых равно NULL
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @return void
 */
public function apply(Builder $builder, Model $model)
{
    $builder->whereNull($model->getQualifiedDeletedAtColumn());

    $this->extend($builder);
}

/**
 * Удаление whereNull
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @return void
 */
public function remove(Builder $builder, Model $model)
{
    $column = $model->getQualifiedDeletedAtColumn();

    $query = $builder->getQuery();

    foreach ((array) $query->wheres as $key => $where)
    {
        // Перебираем wheres в запросе, и когда находим наш - удаляем его из массива wheres по ключу.
        // Это позволяет включать удаленную модель в отношения во время ленивой загрузки.
        if ($this->isSoftDeleteConstraint($where, $column))
        {
            unset($query->wheres[$key]);

            $query->wheres = array_values($query->wheres);
        }
    }
}
```

Отношения

Ваши таблицы скорее всего как-то связаны с другими таблицами БД. Например, статья в блоге может иметь несколько комментариев, а заказ может быть с связан с оставившим его пользователем. Eloquent упрощает работу и управление такими отношениями. Laravel поддерживает несколько типов связей:

- Один к одному
- Один ко многим
- Многие ко многим
- Has Many Through
- Полиморфические связи
- Полиморфические связи многие ко многим

Один к одному

Создание связи «один к одному»

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

Заметьте, что Eloquent считает, что внешнее поле (`foreign_key`) в связанной 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`. Если же в модели, для которой вы строите отношение (в данном случае `User`) ключ находится не в столбце `id`, то вы можете указать его в качесте третьего аргумента:

```
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` используйте 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`. Если вы хотите назвать этот ключ по-другому, передайте это имя вторым параметром к методу `belongsTo`:

```
class Phone extends Model {

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

}
```

Вы можете добавить третий параметр, чтобы указать, какой столбец в родительской модели (`User`) сопоставлять столбцу `local_key`:

```
class Phone extends Model {

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

}
```

Здесь Laravel вернет объекты модели `User`, у которых значение `parent_key` будет равно значению столбца `local_key` у `Phone`.

Один ко многим

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

```
class Post extends Model {

    public function comments()
    {
        return $this->hasMany('Comment');
    }

}
```

Теперь мы можем получить все комментарии с помощью динамического свойства:

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

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

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

Как обычно, вы можете передать второй параметр к методу `hasMany` для перекрытия стандартного имени внешнего ключа в `Comment` (в данном случае по дефолту тут `post_id`) и третий параметр для перекрытия стандатного имени местного ключа ('id' по умолчанию):

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

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

Определение обратного отношения

Для определения обратного отношения используйте метод `belongsTo`:

```
class Comment extends Model {

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

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. Например, один пользователь может иметь роль «Admin». Нужны три таблицы для этой связи: `users`, `roles` и `role_user`. Название таблицы `role_user` происходит от **упорядоченного по алфавиту** имён связанных моделей и 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;
```

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');
    }

}
```

Множество связей через третью таблицу (Has Many Through)

Отношение «has many through» обеспечивает удобный короткий путь для доступа к удаленным отношениям через промежуточные отношения. Например, сделаем так, чтобы модель Country могла иметь много Post **через** модель User. Структура таблиц такая:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Для того, чтобы получить $country->posts, определим вот такое отношение hasManyThrough:

```
class Country extends Model {

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

}
```

Как обычно, если наименование столбцов у вас в таблице отличается от общепринятого, вы можете указать их названия явно:

```
class Country extends Model {

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

}
```

Полиморфические отношения

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

```
class Photo extends Model {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Model {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}

class Order extends Model {

    public function photos()
```

```
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

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

Чтение полиморфической связи

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

Однако истинная «магия» полиморфизма происходит при чтении связи на модели `Photo`:

Чтение связи на владельце полиморфического отношения

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

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

Чтобы понять, как это работает, давайте изучим структуру БД для полиморфического отношения.

Структура таблиц полиморфической связи

```
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 сотрудника, а второе - имя класса-модели владельца. Это позволяет ORM определить, какой класс модели должен быть возвращён при использовании отношения `imageable`.

Полиморфические отношения «многие ко многим»

Помимо стандартных полиморфических отношений, вы можете определить полиморфические отношения «многие ко многим». Например, у нас есть блог, в котором могут публиковаться `Post` и `Video`, и каждый из них может иметь набор тэгов `Tag`.

Структура 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
```

Модели `Post` и `Video` будут иметь отношение `morphToMany`:

```
class Post extends Model {

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

}
```

А модель `Tag` - `morphedByMany` для `Post` и `Video`:

```
class Tag extends Model {

    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }

}
```

Запросы к отношениям

Проверка связей при выборке

При чтении отношений модели вам может быть нужно 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();
```

Вы также можете указать оператор и число:

```
$posts = Post::has('comments', '>=', 3)->get();
```

Метод has работает и с вложенными отношениями:

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

Здесь будут выбраны только те посты, которые имеют комментарии, за которые хотя бы кто-то проголосовал.

Методы whereHas и orWhereHas позволяют использовать where в has-запросах:

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

})->get();
```

Здесь в $posts будут только те посты, у которых есть комменты, начинающиеся на 'foo'.

Динамические свойства

Eloquent позволяет вам читать отношения через динамические свойства. Eloquent автоматически определит используемую связь и даже вызовет get для связей «один ко многим» и first - для связей «один к одному». Например, у нас есть модель 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;
```

> **Примечание:** Отношения, которые возвращают несколько результатов, возвращают коллекцию, т.е. объект Illuminate\Database\Eloquent\Collection

Жадная загрузка (eager loading)

Жадная загрузка (eager loading) призвана устранить проблему 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, у нас есть такой код:

```
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.

К счастью, мы можем использовать жадную загрузку для кардинального уменьшения числа запросов. При вызове метода with отношения будет загружены все за один запрос:

```
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 жадной загрузки может кардинально повысить производительность вашего приложения.

Вы можете загрузить несколько отношений одновременно:

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

Вы даже можете загрузить вложенные отношения:

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

В примере выше, связь author будет загружена вместе со связью contacts модели автора.

Ограничения жадной загрузки

Иногда вам может быть нужно не только 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».

Ленивая жадная загрузка

Можно подгрузить отношения динамически, т.е. после того как вы получили коллекцию основных объектов. Это может быть полезно тогда, когда вам неизвестно, понадобятся вам далее отношения или нет.

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

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

Вы также можете дополнить запрос при помощи функции-замыкания:

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

Вставка связанных моделей

Создание связанной модели

Часто вам нужно будет добавить связанную модель. Например, вы можете добавить новый комментарий к посту. Вместо явного указания значения для поля `post_id` вы можете вставить модель через её родительскую модель - `Post`:

```
$comment = new Comment(['message' => 'Новый комментарий.']);

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

В этом примере поле `post_id` вставленного комментария автоматически получит значение ID поста, к которому он оставлен.

Если вам надо сохранить несколько связанных моделей:

```
$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);
```

Связывание моделей Belongs To

При обновлении связей `belongsTo` («принадлежит к») вы можете использовать метод `associate`. Он установит внешний ключ (foregin key) на связанной модели:

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

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

$user->save();
```

Связывание моделей Many to Many

Вы также можете вставлять связанные модели при работе с отношениями «многие ко многим». Продолжим использовать наши модели `User` и `Role` в качестве примеров (`User` может иметь несколько `Role`). Вы можем легко привязать новые роли к пользователю методом `attach`.

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` - метод `detach`:

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

И `attach` и `detach` могут принимать массивы ID:

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

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

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

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 только переданные ID будут существовать в промежуточной таблице для данной модели.

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

Добавление данных для связующий 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]);
```

Обновление данных времени владельца

Когда модель принадлежит к другой посредством belongsTo - например, Comment, принадлежащий Post- иногда нужно обновить время изменения владельца при обновлении связанной модели. Например, при изменении модели Comment вы можете обновлять поле updated_at её модели Post. Eloquent делает этот процесс простым - просто добавьте свойство touches, содержащее имена всех отношений с моделями-потомками.

```
class Comment extends Model {

    protected $touches = ['post'];

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

Теперь при обновлении Comment владелец Post также обновит своё поле updated_at:

```
$comment = Comment::find(1);

$comment->text = 'Изменение этого комментария.';

$comment->save();
```

Работа со связующими таблицами

Как вы уже узнали, работа отношения многие ко многим требует наличия промежуточной (pivot) таблицы. Например, предположим, что наш объект User имеет множество связанных объектов Role. После чтения отношения мы можем прочитать свойство pivot на обоих 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.

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();
```

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

Удаление всех связующих записей

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

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

Обновление записей в связующей таблице

Иногда вам нужно просто обновить связующую таблицу, не присоединяя или отсоединяя ничего. Используйте метод updateExistingPivot:

```
User::find(1)->roles()->updateExistingPivot($roleId, $attributes);
```

Своя модель Pivot

Laravel позволяет создать свою модель Pivot для работы со связанной таблицей. Сначала создайте модель, которая наследуется (extends) от Eloquent, и назовите её, например, BaseModel. Далее все свои модели наследуйте уже от неё. А в BaseModel создайте следующую функцию, которая возвращает объект для работы со связанной таблицей:

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
}
```

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 наподобие массива. Однако этот объект также имеет набор других полезных методов для работы с результатом запроса.

Например, мы можем выяснить, содержит ли результат запись с определённым первичным ключом методом `contains`.

Проверка на существование ключа в коллекции

```
$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();
```

Если коллекция преобразуется в строку результатом будет JSON-выражение:

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

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

Проход и фильтрация элементов коллекции

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

});

$roles = $user->roles->filter(function($role)
{

});
```

Применение функции обратного вызова

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

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

Сохранение коллекции по значению

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

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

Использование произвольного класса коллекции

```
class User extends Model {

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

}
```

Читатели (accessors) и преобразователи (mutators)

Eloquent содержит мощный механизм для преобразования атрибутов модели при их чтении и записи. Просто объявите в её классе метод `getFooAttribute`. Помните, что имя метода должно следовать соглашению camelCase, даже если поля таблицы используют соглашение snake-case, т.е. с подчёркиваниями.

Объявление читателя

```
class User extends Model {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

В примере выше поле `first_name` теперь имеет читателя (accessor). Заметьте, что оригинальное значение атрибута передаётся методу в виде параметра.

Преобразователи (mutators) объявляются подобным образом.

Объявление преобразователя

```
class User extends Model {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

Преобразователи дат

По умолчанию Eloquent преобразует поля `created_at` и `updated_at` в объекты Carbon, которые предоставляют множество полезных методов, расширяя стандартный класс PHP `DateTime`.

Вы можете указать, какие поля будут автоматически преобразованы и даже полностью отключить преобразование перекрыв метод `getDates` класса модели.

```
public function getDates()
{
    return ['created_at'];
}
```

Когда поле является датой, вы можете установить его в число-оттиск времени формата Unix (timestamp), строку даты формата(`Y-m-d`), строку даты-времени и, конечно, экземпляр объекта `DateTime` или `Carbon`.

Чтобы полностью отключить преобразование дат просто верните пустой массив из метода `getDates`.

```
public function getDates()
{
    return [];
}
```

Приведение атрибутов к определенному типу

Бывает, что некоторые атрибуты модели вам надо приводить к определенному типу данных. Можно, конечно, написать к этим полям преобразователи, а можно поступить проще - установить свойство `casts`, в котором перечислить эти аттрибуты и типы данных, к которым их нужно приводить. Например:

```
/**
 * Теперь атрибут is_admin - логического типа
 *
 * @var array
 */
protected $casts = [
    'is_admin' => 'boolean',
];
```

Несмотря на то, что в БД `is_admin` хранится в целочисленном (integer) поле, в модели он будет хранится в виде логической переменной - `true` или `false`.

Поддерживаемые типы данных: `integer`, `real`, `float`, `double`, `string`, `boolean`, и `array`.

Преобразование `array` может быть особенно полезно для столбцов, в которых хранится сериализованный JSON. Такие столбцы будут автоматически десериализовываться и JSON будет преобразовываться в массив:

```
protected $casts = [
    'options' => 'array',
];
```

Теперь мы можем хранить массивы в БД:

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

// $options - массив, полученный из JSON
$options = $user->options;

// options автоматически сериализуется обратно в JSON !
$user->options = ['foo' => 'bar'];
```

События моделей

Модели Eloquent инициируют несколько событий, что позволяет вам добавить к ним свои обработчики с помощью следующих методов: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`.

Когда первый раз сохраняется новая модель возникают события `creating` и `created`. Если модель уже существовала на момент вызова метода `save`, вызываются события `updating` и `updated`. В обоих случаях также возникнут события `saving` и `saved`.

Отмена сохранения модели через события

Если обработчики `creating`, `updating`, `saving` или `deleting`вернут значение `false`, то действие будет отменено.

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

Где регистрировать слушателей событий

Регистрировать слушателей событий модели можно в вашем сервис-провайдере `EventServiceProvider`. Например:

```
/**
 * Это место для регистрации слушателей событий (event listeners) вашего приложения
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    User::creating(function($user)
    {
        //
    });
}
```

Наблюдатели моделей

Для того, чтобы держать всех обработчиков событий моделей вместе вы можете зарегистрировать наблюдателя (observer). Объект-наблюдатель может содержать методы, соответствующие различным событиям моделей. Например, методы `creating`, `updating` и `saving`, а также любые другие методы, соответствующие именам событий.

К примеру, класс наблюдателя может выглядеть так:

```
class UserObserver {
```

```
    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
        //
    }

}
```

Вы можете зарегистрировать его используя метод observe:

```
User::observe(new UserObserver);
```

Генерация урлов

Вы можете передать экземпляр модель в хелперах route или action, в урл будет вставлен соответствующий ключ. Например:

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

action('UserController@show', [$user]);
```

Здесь в {user} будет передано $user->id. Вы можете изменить передаваемый в урл ключ следующим методом модели:

```
public function getRouteKey()
{
    return $this->slug;
}
```

Преобразование в массивы и JSON

При создании JSON API вам часть потребуется преобразовывать модели к массивам или выражениям JSON. Eloquent содержит методы для выполнения этих задач. Для преобразования модели или загруженного отношения к массиву можно использовать метод toArray.

Преобразование модели к массиву

```
$user = User::with('roles')->first();

return $user->toArray();
```

Заметьте, что целая коллекция моделей также может быть преобразована к массиву:

```
return User::all()->toArray();
```

Для преобразования модели к JSON, вы можете использовать метод toJson:

Преобразование модели к JSON

```
return User::find(1)->toJson();
```

Обратите внимание, что если модель преобразуется к строке, результатом также будет JSON - это значит, что вы можете возвращать объекты Eloquent напрямую из ваших маршрутов!

Возврат модели из маршрута

```
Route::get('users', function()
{
    return User::all();
});
```

Иногда вам может быть нужно ограничить список атрибутов, включённых в преобразованный массив или JSON-строку - например, скрыть пароли. Для этого определите в классе модели свойство hidden.

Скрытие атрибутов при преобразовании в массив или JSON

```
class User extends Model {

    protected $hidden = ['password'];

}
```

Вы также можете использовать атрибут visible для указания разрешённых полей:

```
protected $visible = ['first_name', 'last_name'];
```

Иногда вам может быть нужно добавить поле, которое не существует в таблице. Для этого просто определите для него читателя:

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

Как только вы создали читателя добавьте его имя к свойству-массиву appends класса модели:

```
protected $appends = ['is_admin'];
```

Как только атрибут был добавлен к списку appends, он будет включён в массивы и выражения JSON, образованные от этой модели. Атрибуты в appends подчиняются правилам visible и hidden модели.

Введение

Класс Schema представляет собой независимый от типа БД интерфейс манипулирования таблицами. Он хорошо работает со всеми БД, поддерживаемыми Laravel и предоставляет унифицированный API для любой из этих систем.

Создание и удаление таблиц

Для создания новой таблицы используется метод `Schema::create`:

```
Schema::create('users', function(Blueprint $table)
{
    $table->increments('id');
});
```

Первый параметр метода `create` - имя таблицы, а второй - функция-замыкание, которое получает объект `Blueprint`, использующийся для определения таблицы.

Чтобы переименовать существующую таблицу используется метод `rename`:

```
Schema::rename($from, $to);
```

Для указания иного использующегося подключения к БД используется метод `Schema::connection`:

```
Schema::connection('foo')->create('users', function(Blueprint $table)
{
    $table->increments('id');
});
```

Для удаления таблицы вы можете использовать метод `Schema::drop`:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Добавление полей

Для изменения существующей таблицы используется метод `Schema::table`:

```
Schema::table('users', function(Blueprint $table)
{
    $table->string('email');
});
```

Конструктор таблиц поддерживает различные типы полей:

Команда	Описание
`$table->bigIncrements('id');`	Первичный последовательный (autoincrement) ключ типа BIGINT
`$table->bigInteger('votes');`	Поле BIGINT
`$table->binary('data');`	Поле BLOB
`$table->boolean('confirmed');`	Поле BOOLEAN

`$table->char('name', 4);`	Поле CHAR с указанием длины
`$table->date('created_at');`	Поле DATE
`$table->dateTime('created_at');`	Поле DATETIME
`$table->decimal('amount', 5, 2);`	Поле DECIMAL с параметрами "точность" (общее количество значащих десятичных знаков) и "масштаб" (количество десятичных знаков после запятой)
`$table->double('column', 15, 8);`	Поле DOUBLE (параметры аналогичны полю DECIMAL)
`$table->enum('choices', array('foo', 'bar'));`	Поле ENUM
`$table->float('amount');`	Поле FLOAT
`$table->increments('id');`	Первичный последовательный (autoincrement) ключ
`$table->integer('votes');`	Поле INTEGER
`$table->json('options');`	Текстовое поле для хранения JSON-данных
`$table->longText('description');`	Поле LONGTEXT
`$table->mediumInteger('numbers');`	Поле MEDIUMINT
`$table->mediumText('description');`	Поле MEDIUMTEXT
`$table->morphs('taggable');`	Создается два поля - INTEGER `taggable_id` и STRING `taggable_type`
`$table->nullableTimestamps();`	То же, что и `timestamps()`, но разрешены NULL
`$table->smallInteger('votes');`	Поле SMALLINT
`$table->tinyInteger('numbers');`	Поле TINYINT
`$table->softDeletes();`	Столбец **deleted_at** для реализации псевдоудаления
`$table->string('email');`	Поле VARCHAR
`$table->string('name', 100);`	Поле VARCHAR с заданной длиной
`$table->text('description');`	Поле TEXT
`$table->time('sunrise');`	Поле TIME
`$table->timestamp('added_on');`	Поле TIMESTAMP
`$table->timestamps();`	Столбцы **created_at** и **updated_at**

`$table->rememberToken();`	Столбец `remember_token` VARCHAR(100) NULL у таблицы users для реализации функции "запомнить меня" при логине
`->nullable()`	данное поле может быть NULL
`->default($value)`	установка дефолтного значения поля
`->unsigned()`	запрещаются отрицательные значения

Вставка поля после существующего (в MySQL)

Если вы используете MySQL, то при изменении таблицы вы можете указать, куда именно добавлять новое поле

```
$table->string('name')->after('email');
```

Изменение полей

Иногда возникает необходимость изменить существующее поле. К примеру, увеличить длину строкового поля. В этом поможет метод change. Давайте увеличим длину поля `name` до 50 символов:

```
Schema::table('users', function($table)
{
    $table->string('name', 50)->change();
});
```

Так же можно указать, может ли поле быть NULL:

```
Schema::table('users', function($table)
{
    $table->string('name', 50)->nullable()->change();
});
```

Переименование полей

Для переименования поля можно использовать метод `renameColumn`. Переименование возможно только при подключенном пакете `doctrine/dbal` в `composer.json`.

```
Schema::table('users', function(Blueprint $table)
{
    $table->renameColumn('from', 'to');
});
```

> **Примечание:** переименование полей типа `enum` не поддерживается.

Удаление полей

Для удаления поля можно использовать метод `dropColumn`. Удаление возможно только при подключенном пакете `doctrine/dbal` в `composer.json`.

Удаление одного поля из таблицы:

```
Schema::table('users', function(Blueprint $table)
{
    $table->dropColumn('votes');
});
```

Удаление сразу нескольких полей

```
Schema::table('users', function(Blueprint $table)
{
    $table->dropColumn(array('votes', 'avatar', 'location'));
});
```

Проверка на существование

Проверка существования таблицы

Вы можете легко проверить существование таблицы или поля с помощью методов hasTable и hasColumn.

```
if (Schema::hasTable('users'))
{
    //
}
```

Проверка существования поля

```
if (Schema::hasColumn('users', 'email'))
{
    //
}
```

Добавление индексов

Есть два способа добавлять индексы: вместе с определением полей, либо отдельно:

```
$table->string('email')->unique();
```

```
$table->unique('email');
```

Ниже список всех доступных типов индексов.

Команда	Описание
`$table->primary('id');`	Добавляет первичный ключ
`$table->primary(array('first', 'last'));`	Добавляет составной первичный ключ
`$table->unique('email');`	Добавляет уникальный индекс
`$table->index('state');`	Добавляет простой индекс

Внешние ключи

Laravel поддерживает добавление внешних ключей (foreign key constraints) для таблиц:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')->on('users');
```

В этом примере мы указываем, что поле `user_id` связано с полем `id` таблицы `users`. Обратите внимание, что поле `user_id` должно существовать перед определением ключа.

Вы также можете задать действия, происходящие при обновлении (on update) и добавлении (on delete) записей.

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

Для удаления внешнего ключа используется метод `dropForeign`. Схема именования ключей - та же, что и индексов:

```
$table->dropForeign('posts_user_id_foreign');
```

> **Внимание:** при создании внешнего ключа, указывающего на автоинкрементное (автоувеличивающееся) числовое поле, не забудьте сделать указывающее поле (поле внешнего ключа) типа `unsigned`.

Удаление индексов

Для удаления индекса вы должны указать его имя. По умолчанию Laravel присваивает каждому индексу осознанное имя. Просто объедините имя таблицы, имена всех его полей и добавьте тип индекса. Вот несколько примеров:

Command	Description
`$table->dropPrimary('users_id_primary');`	Удаление первичного ключа из таблицы "users"

`$table->dropUnique('users_email_unique');`	Удаление уникального индекса на полях "email" и "password" из таблицы "users"
`$table->dropIndex('geo_state_index');`	Удаление простого индекса из таблицы "geo"

Удаление столбцов дат создания и псевдоудаления

Для того, чтобы удалить столбцы **created_at** , **updated_at** и **deleted_at**, которые создаются методами `timestamps` (или `nullableTimestamps`) и `softDeletes`, вы можете использовать следующие методы:

Команда	Описания
`$table->dropTimestamps();`	Удаление столбцов **created_at** и **updated_at**
`$table->dropSoftDeletes();`	Удаление столбца **deleted_at**

Storage Engines (системы хранения)

Для задания конкретной системы хранения таблицы установите свойство `engine` объекта конструктора:

```
Schema::create('users', function(Blueprint $table)
{
    $table->engine = 'InnoDB';

    $table->string('email');
});
```

> **Система хранения** - тип архитектуры таблицы. Некоторые СУБД поддерживают только свой встроенный тип (такие, как SQLite), в то время другие - например, MySQL - позволяют использовать различные системы даже внутри одной БД (наиболее используемыми являются `MyISAM`, `InnoDB` и `MEMORY`).

Введение

Миграции - это что-то вроде системы контроля версий для вашей базы данных. Они позволяют команде программистов изменять структуру БД, в то же время оставаясь в курсе изменений других участников. Миграции обычно идут рука об руку с конструктором таблиц для более простого обращения с архитектурой вашего приложения.

Создание миграций

Для создания новой миграции вы можете использовать команду `make:migration`:

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

Миграция будет помещена в папку `database/migrations` и будет содержать текущее время, которое позволяет библиотеке определять порядок применения миграций.

> **Примечание:** Старайтесь давать миграциям многословные имена - например, не `comments`, а `create_comments_table` - так вы избежите возможного конфликта названий классов.

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 create_users_table --table=users --create
```

Применение миграций

Накатывание всех новых неприменённых миграций

```
php artisan migrate
```

> **Внимание:** если при применении миграций вы сталкиваетесь с ошибкой "class not found" ("Класс не найден") - попробуйте выполнить команду `composer dump-autoload`.

Применение миграций на продакшне

Если ошибиться в написании миграций, то можно потерять данные в БД. Поэтому при применении миграций на рабочем сервере (рабочая среда по умолчанию, production) Laravel спрашивает подтверждения операции. Чтобы этого не происходило, например, для применения миграций в автоматическом режиме во время деплоя (загрузки) приложения на сервер, используйте ключ `--force`:

```
php artisan migrate --force
```

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

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 для вас уже определён класс `DatabaseSeeder`. Из этого класса вы можете вызывать метод `call` для подключения других классов с данными, что позволит вам контролировать порядок их

выполнения.

Примерные классы для загрузки начальных данных

```
class DatabaseSeeder extends Seeder {

    public function run()
    {
        $this->call('UserTableSeeder');

        $this->command->info('Таблица пользователей заполнена данными!');
    }

}

class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        User::create(array('email' => 'foo@bar.com'));
    }

}
```

Для добавления данных в БД используйте артизан-команду `db:seed`:

```
php artisan db:seed
```

По умолчанию команда `db:seed` запускает метод `run()` класса `DatabaseSeeder`. В этом методе вы можете вызывать другие ваши сидеры. Или, вы можете задать название класса, который будет вызван вместо дефолтного:

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

Вы также можете заполнить БД первоначальными данными командой `migrate:refresh`, которая перед этим откатит и заново применит все ваши миграции:

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

Введение

Redis - продвинутое хранилище пар ключ/значение. Его часто называют сервисом структур данных, так как ключи могут содержать строки, хэши, списки, наборы, and сортированные наборы.

Прежде чем использовать Redis, необходимо установить пакет `predis/predis` версии `~1.0` через Composer.

> **Внимание:** Если у вас установлено расширение Redis через PECL, вам нужно переименовать псевдоним в файле `config/app.php`.

Настройка

Настройки вашего подключения к Redis хранятся в файле `app/config/database.php`. В нём вы найдёте массив `redis`, содержащий список серверов, используемых приложением:

```
'redis' => [

    'cluster' => true,

    'default' => ['host' => '127.0.0.1', 'port' => 6379],

],
```

Если у вас Redis установлен на других портах, или есть несколько redis-серверов, дайте имя каждому подключению к Redis и укажите серверные хост и порт.

Параметр `cluster` сообщает клиенту Redis Laravel, что нужно выполнить фрагментацию узлов Redis (client-side sharding), что позволит вам обращаться к ним и увеличить доступную RAM. Однако заметьте, что фрагментация не справляется с падениями, поэтому она в основном используется для кэшировании данных, которые доступны из основного источника.

Если ваш сервер Redis требует авторизацию, вы можете указать пароль, добавив к параметрам подключения пару ключ/значение `password`.

Использование

Вы можете получить экземпляр Redis методом `Redis::connection`:

```
$redis = Redis::connection();
```

Так вы получите экземпляр подключения по умолчанию. Если вы не используете фрагментацию, то можно передать этому методу имя сервера для получения конкретного подключения, как оно определено в файле настроек.

```
$redis = Redis::connection('other');
```

Как только у вас есть экземпляр клиента Redis вы можете выполнить любую команду Redis .Laravel использует магические методы PHP для передачи команд на сервер:

```
$redis->set('name', 'Тейлор');

$name = $redis->get('name');

$values = $redis->lrange('names', 5, 10);
```

Как вы видите, параметры команд просто передаются магическому методу. Конечно, вам не обязательно использовать эти методы - вы можете передавать команды на сервер методом `command`:

```
$values = $redis->command('lrange', array(5, 10));
```

Если у вас в конфиге определено одно дефолтное подключение, то вы можете использовать статические методы:

```
Redis::set('name', 'Тейлор');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

> **Примечание:** Laravel поставляется с драйверами Redis для кэширования и сессий.

Конвейер

Конвейер (pipelining) должен использоваться, когда вы отправляете много команд на сервер за одну операцию. Для начала выполните команду `pipeline`:

Отправка конвейером набора команд на сервер

```
Redis::pipeline(function($pipe)
{
    for ($i = 0; $i < 1000; $i++)
    {
        $pipe->set("key:$i", $i);
    }
});
```

Введение

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

Использование

Вывод всех доступных команд

Чтобы вывести все доступные команды Artisan, используйте команду `list`:

```
php artisan list
```

Просмотр помощи для команды

Каждая команда также включает и инструкцию, которая отображает и описывает доступные аргументы и опции для команды. Для того, чтобы её вывести, просто добавьте слово `help` перед командой:

```
php artisan help migrate
```

Запуск в заданной среде выполнения

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

```
php artisan migrate --env=local
```

Отображение используемой версии Laravel

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

```
php artisan --version
```

Вызов команд из приложения

Иногда может потребоваться выполнить команду Artisan из вашего приложения, например, в обработчике роута или в контроллере. Для этого используется фасад `Artisan`:

```
Route::get('/foo', function()
{
    $exitCode = Artisan::call('command:name', ['--option' => 'foo']);

    //
});
```

Вы даже можете добавить команду в очередь для того, чтобы она выполнялась на фоне менеджером очереди:

```
Route::get('/foo', function()
{
    Artisan::queue('command:name', ['--option' => 'foo']);

    //
});
```

Планировщик заданий

Раньше разработчикам приходилось добавлять задание в Cron для каждой консольной команды и это была большая головная боль. Давайте сделаем нашу жизнь проще. Планировщик заданий Laravel позволяет просто и гибко составлять расписание запуска ваших команд из самого приложения и для этого потребуется добавить всего одно Cron задание.

Ваш планировщик находится в файле `app/Console/Kernel.php`. В классе `Kernel` вы увидите метод `schedule`, который уже содержит в себе простой пример. Вы можете добавить сколько угодно заданий, используя объект `Schedule`. Единственное Cron задание, которое нужно добавить на сервер:

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

Это Cron задание вызывает планировщик заданий каждую минуту. Затем, Laravel просматривает задания и запускает необходимые. Проще некуда!

Несколько примеров

Давайте рассмотрим несколько примеров использования планировщика:

Замыкание в качестве задания

```
$schedule->call(function()
{
    // Do some task...

})->hourly();
```

Консольная команда в качестве задания

```
$schedule->exec('composer self-update')->daily();
```

Добавление задания, используя синтаксис Cron

```
$schedule->command('foo')->cron('* * * * *');
```

Постоянные задания

```
$schedule->command('foo')->everyFiveMinutes();

$schedule->command('foo')->everyTenMinutes();

$schedule->command('foo')->everyThirtyMinutes();
```

Ежедневные задания

```
$schedule->command('foo')->daily();
```

Ежедневные задания с запуском в определённое время (24-часовой формат времени)

```
$schedule->command('foo')->dailyAt('15:00');
```

Задания, выполняемые дважды в день

```
$schedule->command('foo')->twiceDaily();
```

Задания на каждый день, кроме выходных

```
$schedule->command('foo')->weekdays();
```

Еженедельные задания

```
$schedule->command('foo')->weekly();

// Можно указать время выполнения для каждого дня (0-6)...
$schedule->command('foo')->weeklyOn(1, '8:00');
```

Ежемесячные задания

```
$schedule->command('foo')->monthly();
```

Запуск по дням недели

```
$schedule->command('foo')->mondays();
$schedule->command('foo')->tuesdays();
$schedule->command('foo')->wednesdays();
$schedule->command('foo')->thursdays();
$schedule->command('foo')->fridays();
$schedule->command('foo')->saturdays();
$schedule->command('foo')->sundays();
```

Выполнение задания только в определённой среде выполнения

```
$schedule->command('foo')->monthly()->environments('production');
```

Выполнение задания, даже если приложение находится в режиме обслуживания

```
$schedule->command('foo')->monthly()->evenInMaintenanceMode();
```

Выполнять, но только если функция-параметр вернула `true`

```
$schedule->command('foo')->monthly()->when(function()
{
    return true;
});
```

Отправить вывод на email

```
$schedule->command('foo')->emailOutputTo('foo@example.com');
```

Записать вывод в файл

```
$schedule->command('foo')->sendOutputTo($filePath);
```

Дернуть url по завершении задачи

```
$schedule->command('foo')->thenPing($url);
```

Введение

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

Создание команды

Генерация класса

Для создания новой команды, вы можете воспользоваться командой Artisan `make:console`, которая сгенерирует макет класса:

Сгенерируйте новый класс команды

```
php artisan make:console Foo
```

Команда выше сгенерирует класс `app/Console/Commands/Foo.php`.

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

```
php artisan make:console AssignUsers --command=users:assign
```

Написание команды

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

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

Аргументы и опции

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

Массив, определяющий аргументы, выглядит так:

```
[$name, $mode, $description, $defaultValue
```

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

Массив, определяющий опции, выглядит следующим образом:

```
[$name, $shortcut, $mode, $description, $defaultValue]
```

Для опций, аргумент `mode` может быть: `InputOption::VALUE_REQUIRED` (значение обязательно), `InputOption::VALUE_OPTIONAL` (значение необязательно), `InputOption::VALUE_IS_ARRAY` (значение - массив), `InputOption::VALUE_NONE` (нет значения).

Режим `VALUE_IS_ARRAY` обозначает, что этот переключатель может быть использован несколько раз при вызове команды:

```
php artisan foo --option=bar --option=baz
```

Значение `VALUE_NONE` означает, что опция просто используется как "переключатель":

```
php artisan foo --option
```

Получение ввода

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

Получение значения аргумента команды

```
$value = $this->argument('name');
```

Получение всех аргументов

```
$arguments = $this->argument();
```

Получение значения опции команды

```
$value = $this->option('name');
```

Получение всех опций

```
$options = $this->option();
```

Вывод команды

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

Вывод информации в консоль

```
$this->info('Display this on the screen');
```

Вывод сообщений об ошибке в консоль

```
$this->error('Something went wrong!');
```

Взаимодействие с пользователем

Вы также можете воспользоваться методами `ask` и `confirm` для обеспечения пользовательского ввода:

Попросить пользователя ввести данные:

```
$name = $this->ask('What is your name?');
```

Попросить пользователя ввести секретные данные:

```
$password = $this->secret('What is the password?');
```

Попросить пользователя подтвердить что-то:

```
if ($this->confirm('Do you wish to continue? [yes|no]'))
{
    //
}
```

Вы также можете указать ответ по умолчанию для метода `confirm`. Это должно быть `true` или `false`:

```
$this->confirm($question, true);
```

Вызов других команд

Иногда может потребоваться вызвать другую команду из вашей команды. Это можно сделать используя метод `call`:

```
$this->call('command:name', ['argument' => 'foo', '--option' => 'bar']);
```

Регистрация команд

Регистрация Artisan-команд

Как только ваша команда будет готова, вам необходимо зарегистрировать её с помощью Artisan CLI, чтобы она была доступна для использования. Это делается в файле `app/Console/Kernel.php`, в свойстве `commands`:

```
protected $commands = [
    'App\Console\Commands\Inspire',
];
```

Для регистрации вашей команды просто добавьте её в этот массив. Когда Artisan загрузится, все команды из этого массива будут автоматически определены с помощь IoC-контейнера и зарегистрированы.