Laravel 4.1

====

Документация на русском языке. Перевод официальной документации. Перевод взят с гитхаба LaravelRUS

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

Установка

Установка Composer

Laravel использует Composer для управления зависимостями. Для начала скачайте файл `composer.phar`. Дальше вы можете либо оставить этот Phar-архив в своей локальной папке с проектом, либо переместить его в `usr/local/bin`, чтобы использовать его в рамках всей системы. Для Windows вы можете использовать официальный установщик.

Установка Laravel

При помощи установщика Laravel

Загрузите Laravel installer PHAR archive. Если у вас Mac OS или Linux, переименуйте его в `laravel` и положите в `/usr/local/bin`. Если у вас Windows, положите `laravel.phar` в папку, которая у вас прописана в PATH, плюс в этой же папке создайте файл `laravel.bat` со следующим содержимым:

```
@php "%~dp0laravel.phar" %*
```

Теперь, к примеру, если вы исполните в терминале команду `laravel new blog`, будет создана папка `blog`, в которую будет загружен фреймворк с уже подтянутыми зависимостями composer. Этот способ установки Laravel наиболее быстрый, особенно на Windows, где Composer работает довольно медленно.

При помощи Composer

Вы можете установить Laravel с помощью команды `create-project`:

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

Загрузка архива

Скачайте последнюю версию фреймворка и извлеките архив в папку на вашем сервере. Далее скачайте Composer в эту же папку и выполните в этой папке `php composer.phar install` (или `composer install`, если composer у вас уже установлен в системе глобально) для установки всех зависимостей библиотеки. Этот процесс также требует, чтобы у вас был установлен Git.

Если вы хотите обновить Laravel выполните команду `php composer.phar update`.

Требования к серверу

У Laravel всего несколько требований к вашему серверу:

- PHP >= 5.3.7
- MCrypt PHP Extension

Настройка

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

После вам будет необходимо определить среду выполнения и отредактировать конфиг-файлы фреймворка - например, разрешить отображение ошибок для вашей среды выполнения (параметр debug в конфиге app.php).

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

Права доступа

Laravel требует, чтобы у сервера были права на запись в папку `app/storage`.

Пути

Некоторые системные пути Laravel - настраиваемые; для этого обратитесь к файлу `bootstrap/paths.php`.

Примечание: Laravel спроектирован так, чтобы защитить код вашего приложения и локальное хранилище - для этого общедоступные файлы помещаются в папку `public`. Подразумевается, что эта папка является корневой папкой вашего сайта (DocumentRoot в Apache).

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 качествет веб-сервера Nginx, то используйте для ЧПУ следующую конструкцию:

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

Настройка

Вступление

Все файлы настроек Laravel хранятся в папке `app/config`. Каждая настройка задокументирована, поэтому не стесняйтесь изучить эти файлы и познакомиться с возможностями конфигурирования.

Иногда вам нужно прочитать настройку во время работы приложения. Это можно сделать, используя класс `Config`:

Чтение значения настройки

```
Config::get('app.timezone');
```

Вы можете указать значение по умолчанию, которое будет возвращено, если настройка не существует:

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

Заметьте, что синтаксис с точкой может использоваться для доступа к разным файлам настроек.

Изменение значения во время выполнения

```
Config::set('database.default', 'sqlite');
```

Значения, установленные таким образом, сохраняются только для текущего запроса и не влияют на более поздние запросы.

Настройки среды выполнения (the environment)

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 продакшн-серверах. Это легко достигается использованием настроек, зависящих от среды.

Просто создайте внутри `config` папку с именем вашей среды, таким как `local`. Затем создайте файлы настроек и укажите в них значения для этой среды, которыми вы перекроете изначальные настройки. Например, вы можете перекрыть драйвер кэша для локальной системы, создав файл `cache.php` внутри `app/config/local` с таким содержимым:

```
<?php

return array(

    'driver' => 'file',

);
```

> **Примечание:** Не используйте имя 'testing' для названия среды - оно зарезервировано для юнит-тестов.

Заметьте, что вам не нужно указывать *каждую* настройку, которая есть в конфигурационном файле по умолчанию (`app/config/cache.php`). Настройки среды будут наложены на эти базовые файлы.

Теперь нам нужно сообщить Laravel, в какой среде он работает. По умолчанию это всегда `production`. Вы можете настроить другие среды в файле `bootstrap/start.php` который находится в корне установки Laravel. В этом файле есть вызов `$app->detectEnvironment` - массив, который ему передаётся, используется для определения текущей среды. Вы можете добавить в него другие среды и имена компьютеров по необходимости.

```
<?php

$env = $app->detectEnvironment(array(

    'local' => array('your-machine-name'),

));
```

В этом примере 'local' это название среды, а 'your-machine-name' - это имя вашего компьютера. Чтобы узнать его, в Mac OS или Linux вы можете использовать команду hostname в терминале, а в Windows - зайти в Панель управления -

Система.

Если вам нужна бОльшая гибкость в определении среды, Вы можете передать в метод функцию-замыкание (Closure) и задавать среду так, как нужно именно вам. Например, если вы хотите задавать среду в файле `.htaccess` , то функция-замыкание будет выглядеть следующим образом:

```
$env = $app->detectEnvironment(function()
{
    return getenv('MY_LARAVEL_ENV');
});
```

При этом в `public/.htaccess` вы добавляете строку с именем среды:

```
SetEnv MY_LARAVEL_ENV local
```

Или, если у вас в качестве веб-сервера nginx, добавляем в правило обработки php-файлов:

```
fastcgi_param MY_LARAVEL_ENV "local";
```

Получение имени текущей среды выполнения

Вы можете получить имя текущей среды с помощью метода `environment`:

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

Для определения среды вы можете передать аргументы в этот метод:

```
if (App::environment('local'))
{
    // Среда - local
}

if (App::environment('local', 'staging'))
{
    // Среда - local ИЛИ staging
}
```

Дополнительные сервис-провайдеры

Иногда требуется для определенной среды выполнения подгрузить дополнительные сервис-провайдеры. Для этого в конфиг-файле app определенной среды выполнения, например, `config/local/app.php`, используйте хелпер `append_config`:

```
'providers' => append_config(array(
    'LocalOnlyServiceProvider',
))
```

Конфиденциальные конфиги

Часто вам может понадобиться хранить часть конфигурационной информации вне обычных конфигов Laravel. Это могут быть ключи доступа к сторонним API-сервисам, ключи шифрования, логины/пароли для доступа к базам данных - или еще какие-то вещи, которые нужны вам только на конкретно этой машине и вы не хотите, чтобы они светились в репозитории. Laravel предоставляет простое средство для размещения таких конфиденциальных конфигов - хранить их в dot-файлах (имена этих файлов начинаются с точки).

Для этого, во-первых, задайте для данного сервера нужную среду выполнения. Например, это ваша девелоперская машина и среда называется local. Затем в корне проекта (там, где лежит `composer.json`) создайте файл `.env.local.php`. Этот файл должен возвращать массив значений, так и стандартный конфиг Laravel:

```
<?php

return array(

    'TEST_STRIPE_KEY' => 'super-secret-sauce',

);
```

Эти значения будут доступны в вашем приложении (например, в конфигах) в суперглобальных переменных `$_ENV` и `$_SERVER`:

```
'key' => $_ENV['TEST_STRIPE_KEY']
```

Для продакшн-сервера файл может называться `.env.php` , а может содержать объявленное в `bootstrap/start.php`

название среды выполнения в своем имени - `.env.production.php`

Теперь вы можете поместить `.env.local.php` в `.gitignore` и держать там, например, ваши личные и особенные логины и пароли и название базы данных, не боясь, что после того, как вы сделаете `git push`, вашим коллегам придется править конфиги, чтобы приложение заработало у них.

Режим обслуживания

Когда ваше приложение находится в режиме обслуживания (maintenance mode), специальный шаблон будет отображаться вместо всех ваших маршрутов. Это позволяет "отключать" приложение в момент обновления кода или обслуживания базы данных. Вызов `App::down`, который задает поведение приложения в этом режиме, уже содержится в файле `app/start/global.php`. Возвращённое им значение будет отправлено пользователю, когда приложение находится в режиме обслуживания.

Для включения этого режима просто выполните команду down Artisan:

```
php artisan down
```

Чтобы выйти из режима обслуживания выполните команду up:

```
php artisan up
```

Для отображения собственного шаблона в режиме обслуживания вы можете добавить в `app/start/global.php` подобный код:

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

Если функция-замыкание вернет NULL, то режим обслуживания не будет включен и приложение будет работать в обычном режиме.

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.

Laravel Homestead

Введение

Laravel старается сделать разработку на PHP восхитительной, включая Вашу локальную среду разработки. Vagrant предоставляет простой и элегантный способ управления и снабжения Виртуальных Машин.

Laravel Homestead является официальным пред-упакованным "боксом" (box) для Vagrant'а, и предоставляет замечательную среду разработки, не требуя от Вас 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 5.5, MySQL, Postgres, Redis, Memcached и другие вкусности, которые могут потребоваться вам для разработки потрясающих Laravel-приложений.

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

Включенное Программное Обеспечение (ПО)

- Ubuntu 14.04
- PHP 5.5
- 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
```

Склонируйте Репозиторий Homestead

Как только бокс будет добавлен в Vagrant, Вам следует склонировать или скачать этот репозиторий. Склонировать репозиторий рекомендуется в некоторую центральную директорию `Homestead`, в которой вы будете хранить все ваши проекты на Laravel, так как Homestead будет выполнять роль управляющего всеми вашими проектами на Laravel (и PHP).

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

Настройте Пути к Вашим SSH ключам

Следующим шагом Вам следует отредактировать файл `Homestead.yaml`, включенный в репозиторий. В этом файле вы можете указать путь к вашему публичному SSH-ключу, а также сконфигурировать совместно используемые Homestead'ом и виртульной машиной папки.

Еще нет SSH ключа? Под Mac и Linux Вы можете создать пару ключей, используя следующую команду:

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

Под Windows вы можете установить Git и воспользоваться командной строкой `Git Bash`, поставляемой с Git, для выполнения этой команды. В качестве альтернативы, Вы можете использовать PuTTY или PuTTYgen.

Как только ключ будет создан, укажите путь к нему в свойстве `authorize` файла `Homestead.yaml`.

Настройте Общедоступные Папки

В свойстве `folders` в файле `Homestead.yaml` перечислены все общедоступные локальные папки, доступ к которым Вы хотите предоставить в среде Homestead. По мере изменения файлов в этих папках, они будут синхронизироваться между локальной машиной и средой Homestead. Настроить можно столько папок, сколько необходимо!

Настройте Nginx Под Ваши Сайты

Еще не знакомы с Nginx? Никаких проблем. Свойство `sites` позволяет легко связать "домен" и произвольную папку в среде Homestead. В файле `Homestead.yaml` имеется пример настрйки одного сервера. Опять же, вы можете добавить столько сайтов, сколько Вам нужно. Homestead может служить удобной виртуальной средой для любого вашего проекта на Laravel!

Алиасы (Aliases) Bash

Чтобы добавить произвольный алиас в Homestead, просто добавьте его в файл `aliases` в корне Homestead.

Запуск Vagrant Бокса

Как только Вы отредактировали файл `Homestead.yaml` по вашему вкусу, выполните команду `vagrant up` из командной строки, находясь в директории Homestead. Vagrant запустит виртуальную машину и настроит все ваши совместно используемые папки и сайты автоматически!

Но не забудьте добавить все ваши "домены" в файл `hosts` в вашей системе! Файл `hosts` будет перенаправлять ваши запросы к локальным доменам в среду Homestead. Под Mac и Linux этот файл находится в `/etc/hosts`. Под Windows он находится здесь: `C:\Windows\System32\drivers\etc\hosts`. Строки, добавляемые Вами в этот файл, будут выглядеть примерно так:

```
127.0.0.1  homestead.app
```

После того, как вы добавите домен в файл `hosts`, Вы получите доступ к сайту из вашего броузера по порту 8000!

```
http://homestead.app:8000
```

Чтобы узнать, как подключаться к Вашим базам данных, читайте дальше!

Повседневное Использование

Соединение По SSH

Чтобы подсоединиться к Вашей среде Homestead по SSH, следует подключиться к `127.0.0.1` по 2222 порту, используя SSH-ключ, указанный в файле `Homestead.yaml`. Вы также можете просто выполнить команду `vagrant ssh`, находясь в директории Homestead.

Если Вам нужно еще большее удобство, может быть полезнымм добавить следующий алиас в файл `~/.bash_aliases` или `~/.bash_profile`:

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

Установка Связи с Базами Данных

База данных `homestead` "из коробки" настроена на использование как MySQL, так и Postgres. Для даже большего удобства, по умолчанию конфигурация базы данных `local` настроена на использование этой базы данных.

Для соединения с Вашей базой данных MySQL или Postgres из вашей основной системы посредством Navicat или Sequel Pro, следует устанавливать соединение с `127.0.0.1` по порту 33060 (MySQL) или 54320 (Postgres). Имя пользователя и пароль для базы данных соответственно `homestead` и `secret`.

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

Создание Дополнительных Сайтов

После того, как среда Homestead будет снабжена необходимым программным обеспеччением и запущена, Вам возможно потребуется добавить дополнительные сайты для ваших Laravel-приложений. В одной среде Homestead может выполяться сколько угодно копий Laravel. Для этого есть два способа. Во-первых, вы можете просто добавить сайты в файл `Homestead.yaml`, после чего выполнить `vagrant destroy` и снова `vagrant up`.

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 домеен в файл `hosts` в вашей основной системе.

Порты

Следующие порты будут перенаправлены в среду 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

Жизненный цикл запроса

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 куда 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. В дополнение к этому мы рассмотрим старт-файлы и события приложения (application events).

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 этого мануала.

Жизненный цикл запроса

При помощи `.htaccess` Апача или правил Nginx или другого веб-сервера, все запросы передаются файлу `public/index.php`. Отсюда Laravel начинает процесс обработки запроса и здесь же он возвращает ответ пользователю.

Безусловно, наиболее важным понятием при изучении процесса начальной загрузки Laravel является **Service Provider** (дословно - "поставщик услуг", далее - "сервис-провайдер"). Вы можете найти список сервис-провайдеров в файле `config/app.php`, в массиве `providers`. Эти провайдеры - основной механизм настройки (bootstraping) функционала Laravel. Но прежде чем подробно разбираться в их работе, давайте вернемся к `public/index.php`. После его вызова загружается файл `bootstrap/start.php`, в котором создается объект `Application`, который также служит в качестве IoC-контейнера.

После создания объекта `Application` устанавливаются пути до некоторых важных папок фреймворка и устанавливаются критерии настроек среды. Затем вызывается скрипт настройки фреймворка, который помимо установки таймзоны, уровня error_reporting и т.п. делает очень важную вещь - регистрацию сервис-провайдеров, объявленных в `config/app.php`.

Простые сервис-провайдеры имеют только один метод: `register()`. Этот метод вызывается, когда сервис-провайдер регистрируется в объекте `Application`. В рамка этого метода сервис-провайдеры регистрируют некоторые свои вещи с IoC-контейнером. По сути, каждый сервис-провайдер как поставщик услуг добавляет одну или несколько функций-замыканий в контейнер, который позволит вам получить доступ к этим "услугам" в вашем приложении. Например, QueueServiceProvider регистрирует функции-замыкания, которые ресолвят различные классы, связанные с очередями. Конечно, сервис-провайдеры могут быть использованы для любых действий по настройке фреймворка, не только для регистрации вещей с IoC-контейнером. Сервис-провайдер может регистрировать слушателей событий (event-listeners), составителей вида (view composers), Artisan-команд и т.п.

После того как все сервис-провайдеры зарегистрированы, загружаются файлы из `app/start`. Затем загружается `app/routes.php`, и исходя из того, какой роут выбирается для работы, объект `Request` отправляется в `Application`.

Суммируя:

1. Запрос от клиента приходит на `public/index.php`.
2. `bootstrap/start.php` создает объект Application и определяет среду выполнения.
3. Внутренний файл `vendor/laravel/framework/src/Illuminate/Foundation/start.php` читает и применяет конфиги и регистрирует сервис-провайдеры.
4. Загружаются файлы в `app/start`.
5. Загружается `app/routes.php`.
6. Объект `Request` отправляется в `Application`, возвращается объект `Response`.
7. Объект `Response` отправляется клиенту.

Теперь посмотрим повнимательнее на файлы в `app/start`.

Старт-файлы

Старт-файлы вашего приложения находятся в папке `app/start`. По умолчанию их там три: `artisan.php`, `global.php` и `local.php`.

О файле `artisan.php` вы можете узнать в секции Artisan.

Файл `global.php`, в частности, содержит регистрацию Logger и подсоединение файла фильтров `app/filters.php`. Вы можете добавить в него что захотите - этот файл вызывается при каждом запросе, независимо от текущей среды выполнения. Файл `local.php`, как следует из названия, подключается только тогда, когда приложение работает в среде выполнения `local`. Больше о средах выполнения и их конфигурирования вы можете узнать в соответствующей секции - Configuration.

Если у вас в приложении есть еще несколько сред выполнения, например `production`, вы можете создать файл `production.php`. Он подключится, когда приложение будет вызвано в соответствующей среде.

Что размещать в старт-файлах

Старт-файлы - это место для кода, инициализирующего (bootstraping) ваше приложение. Например, вы можете зарегистрировать там view composers, задать настройки логирования, установить некоторые настройки PHP, которые нужны вашему приложению. Конечно, помещать абсолютно весь инициализирующий код в старт-файлы не стоит, так как в этом случае в этом коде легко запутаться. Если вы понимаете, что в старт-файлах у вас помойка, выделяйте часть инициализирующего кода в сервис-провайдеры.

События приложения

Регистрация обработчиков событий

Вы также можете делать пред- или пост-обработку запроса, регистрируя обработчики событий `before`, `after`, `close`, `finish` и `shutdown`:

```
App::before(function()
{
    //
});

App::after(function($request, $response)
{
    //
});
```

Эти обработчики будут запущены соответственно до и после каждого вызова в вашем приложении. Эти события полезны, если вам надо фильтровать запрос или модифицировать ответ глобально для всего приложения. Вы можете зарегистрировать эти обработчики в глобальном старт-файле или в одном из ваших сервис-провайдеров.

Вы также можете зарегистрировать событие, которое вызовется по совпадению роута (маршрута). Событие запускается перед запуском роута.

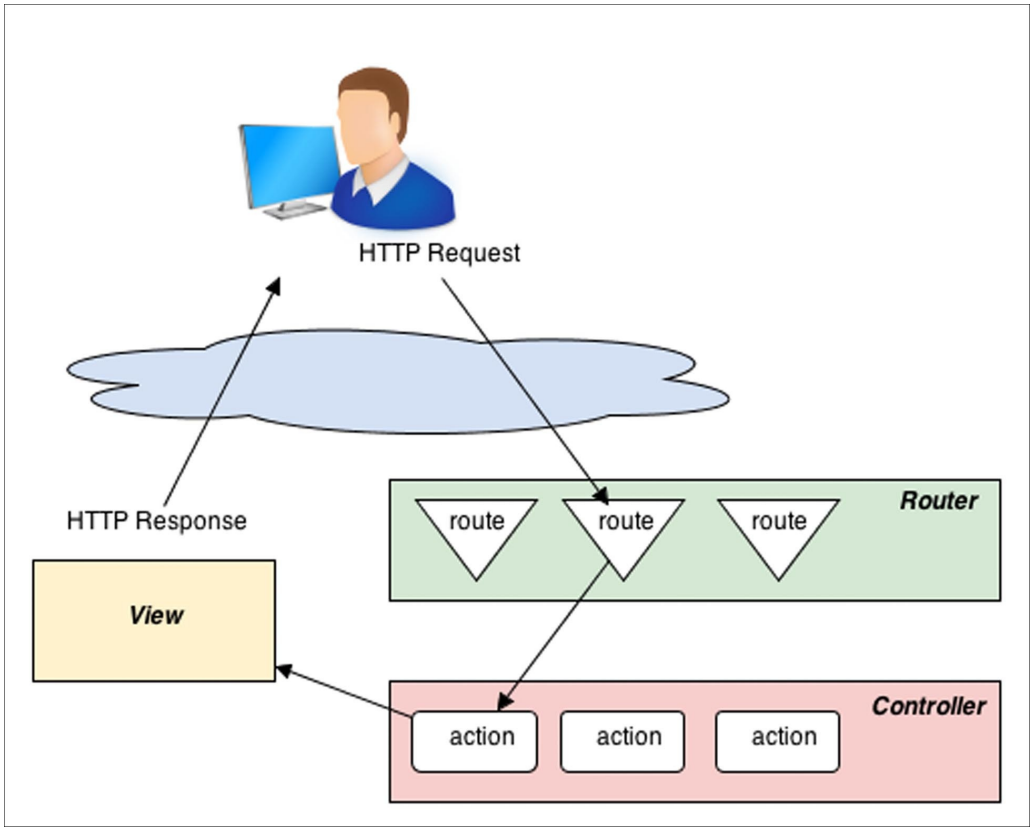
```
Route::matched(function($route, $request)
{
    //
});
```

Событие `finish` вызывается после того, как ваше приложение отсылает сформированный ответ клиенту. Событие `shutdown` вызывается немедленно после всех обработчиков события `finish` и это последняя возможность сделать какие-то действия перед тем как приложение завершится. Скорее всего у вас не будет необходимости использовать эти события.

Приложение 1. Жизненный цикл запроса в деталях

Оригинал: http://laravel-recipes.com/recipes/52

Стандартный жизненный цикл:



Стандартный жизненный цикл состоит из следующих пунктов:

1. HTTP-запрос через Роуты (Routes) поступает в Контроллер (Controller)
2. Контроллер осуществляет некоторые действия в зависимости от запроса и передает данные во Отображения (Views)
3. Отображения отображают полученные данные заданным образом, обеспечивая HTTP-ответ.

Есть много отклонений и различных вариантов вышеприведенной схемы, но она дает нам три опорные точки, на которые надо обратить внимание:

1. Роуты - `app/routes.php`
2. Контроллеры - `app/controllers/`
3. Отображения - `app/views/`

"Отклонения" могут быть, например, такими:

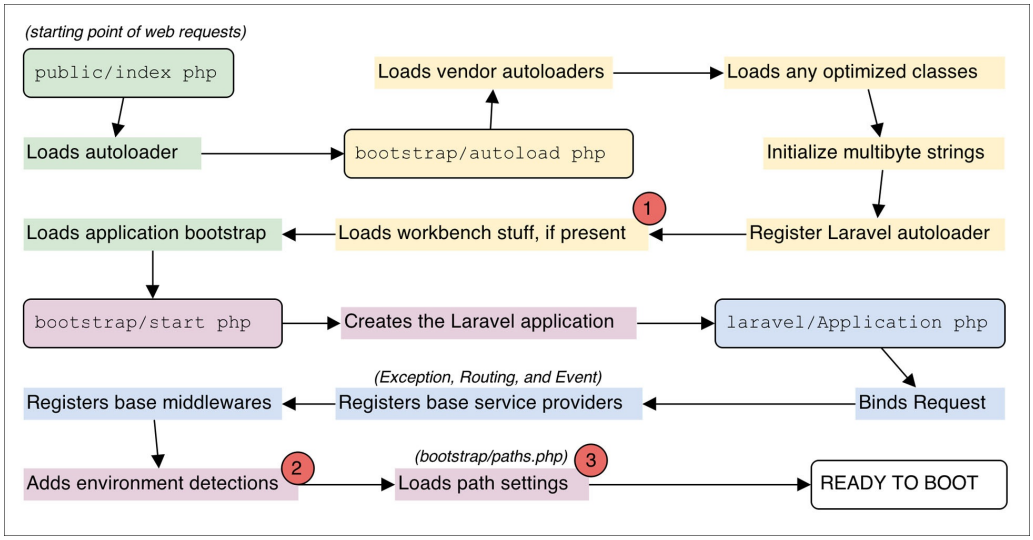
1. Роуты могут возвращать Отображения или сам Ответ (объект `Response`), без задействования Контроллеров.
2. До или после Роутов могут срабатывать Фильтры (`app/filters.php`)
3. В процесс могут вмешаться Исключения (Exceptions) или ошибки приложения.
4. Отклики на события.

Копнем глубже

Более глубокое понимание жизненного цикла запроса в Laravel позволит вам понять, где именно можно (и стоит) писать ваш код.

Цикл запроса можно разбить на три части: **Загрузка** (Loading), **Инициализация** (Booting) и **Выполнение** (Running).

Загрузка (Loading)



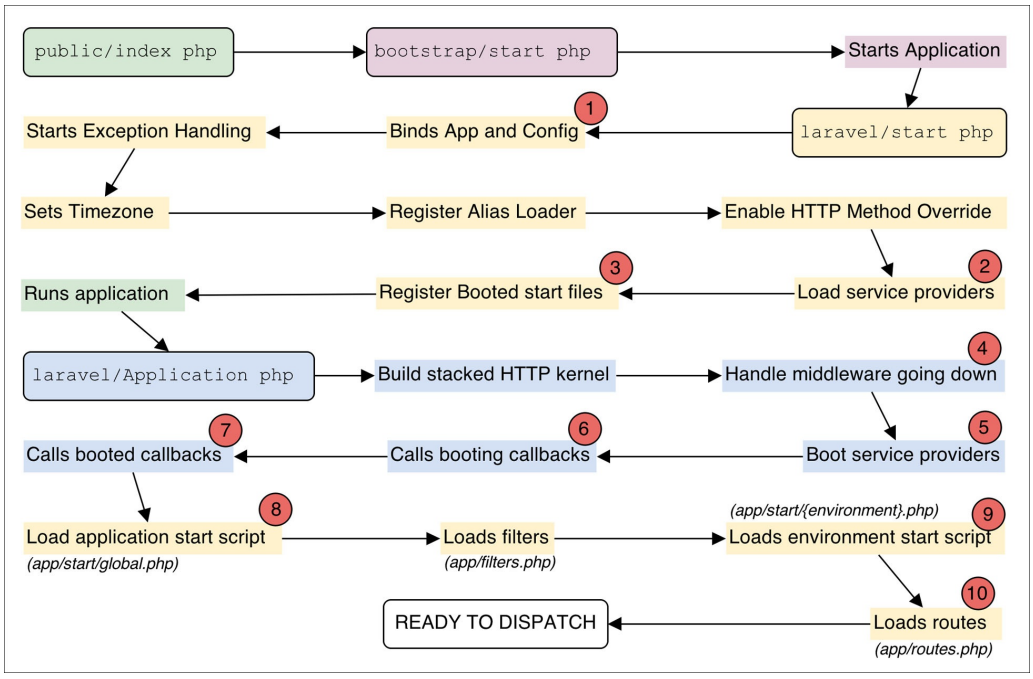
Вот три основные области, где ваше приложение может повлиять на процесс загрузки фреймворка:

1. Пакеты Workbench. Workbench - это способ организовывать свой код в обособленные пакеты и тестировать их внутри вашего приложения перед тем как сделать их пакетами, распространяемыми через Composer. См. Workbench

2. Среда выполнения. В зависимости от установки среды, будут загружены те или иные конфиги, те или иные старт -файлы.

3. Пути. Редактируя `bootstrap/paths.php` вы можете изменить файловую структуру фреймворка, расположив файлы в удобных для вас местах.

Инициализация (Booting)



Есть 10 областей, гда вы можете влиять на процесс инициализации фреймворка.

1. Конфиги. Конфиги влияют и на процесс инициализации и на процесс работы фреймворка.

2. Сервис-провайдеры (Service Providers) Любые сервис-провайдеры, которые вы создали или подсоединили к своему приложению в конфиге `config/app.php` загружаются в начале процесса инициализации. Если сервис-провайдер не отложенный, вызывается его метод `register()`.

3. Загрузка и применение старт-файлов. Регистрируются старт-файлы, которые надо загрузить, когда будет вызвано событие `booted`.

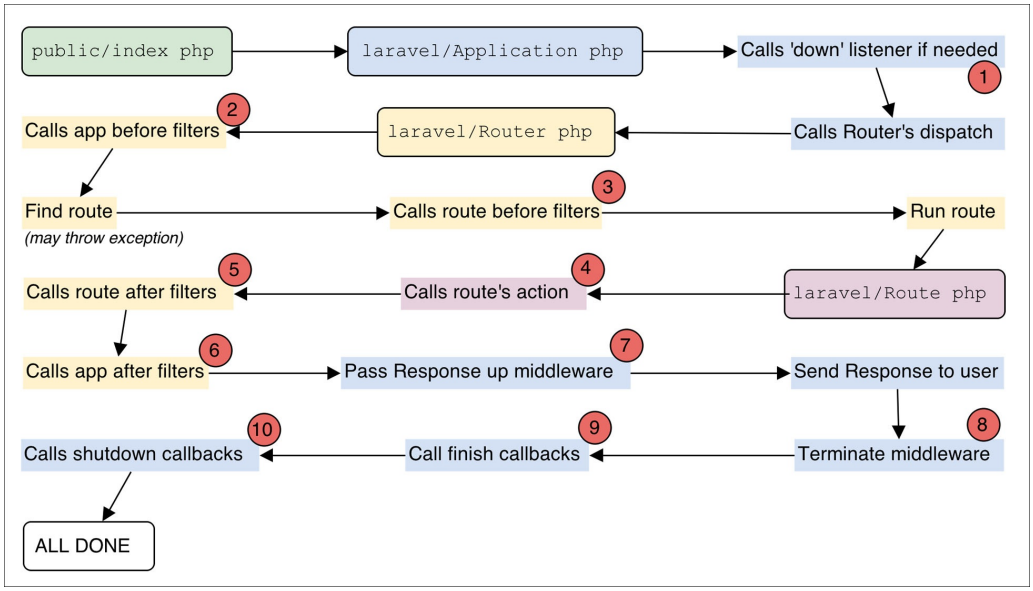
4. Стэк middleware разворачивается вниз Middleware вложены один в другой как матрешки. Верхний middeware обрабатывает запрос и вызывает middleware следующего уровня, и так далее. Последний middleware вызывает приложение. Все middlewares заносятся в стек, и будут вызваны снова в конце части Выполнения (Running).

5. Инициализация сервис-провайдеров Вызывается метод `boot()` у всех зарегистрированных не-отложенных

сервис-провайдеров.

6. Коллбэки пре-инициализации Вызываются все функции-замыкания, зарегистрированные в `App::booting()`
7. Коллбэки пост-инициализации Вызываются все функции-замыкания, зарегистрированные в `App::booted()`. Загружаются старт-файлы, зарегистрированные на шаге 3.
8. Глобальные старт-файлы. Первым делом это `app/start/global.php`, затем, если исполняется artisan-команда, то `app/start/artisan.php`.
9. Старт-файл среды выполнения. Исполняется файл, который имеет то же имя файла, что и название среды выполнения - `app/start/{environment}.php
10. Роуты Исполняется `app/routes.php`. Этот файл вы будете редактировать наиболее часто в процессе разработки вашего приложения.

Выполнение (Running)



10 областей, где вы можете влиять на процесс выполнения:

1. Режим обслуживания Если вы зарегистрировали функцию-подписчика режима обслуживания и приложение находится в этом режиме, эта функция выполняется.
2. Фильтр `before` уровня приложения Если у вас есть фильтры, зарегистрированные в `App::before()`, они выполняются.
3. Фильтры `before` в роутах. Если у вас есть фильтры `before` в роутах, они выполняются.
4. Исполнение запроса. После разбора, к какому роуту относится запрос, вызывается экшн нужного контроллера или коллбэк роута.
5. Фильтры `after` в роутах. Если у вас есть фильтры `after` в роутах, они выполняются.
6. Фильтр `after` уровня приложения Если у вас есть фильтры, зарегистрированные в App::after(), они выполняются .
7. Стек middleware разворачивается вверх Это точка, где объект `Response` передается вверх по стеку `middlewares`. Каждый `middleware` может изменять этот объект.
8. Middleware `shutdown`. Если у вас есть `middleware`, которые реализуют `TerminableInterface`, вызывается метод `shutdown()` этих `middleware`.
9. Коллбэки `finish` Если у вас есть функции, зарегистрированные в `App::finish()`, они выполняются.
10. Коллбэки `shutdown` Если у вас есть функции, зарегистрированные в `App::shutdown()`, они выполняются.

Роуты (маршрутизация)

Простейшая маршрутизация

Большинство роутов (маршруты, routes) вашего приложения будут определены в файле `app/routes.php`. В Laravel простейший роут состоит из URI (урла, пути) и функции-замыкания (она же коллбек).

Простейший GET-роут:

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

Простейший POST-роут:

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

Регистрация роута для нескольких методов

```
Route::match(array('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::to('foo');
```

Здесь 'foo' - это URI.

Параметры роутов

```
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(array('id' => '[0-9]+', 'name' => '[a-z]+'))
```

Определение глобальных паттернов

Вы можете определить соответствие параметра пути регулярному выражению глобально для всех роутов. Например, если у вас `{id}` в урлах это всегда числовое значение:

```
Route::pattern('id', '[0-9]+');

Route::get('user/{id}', function($id)
{
    // Only called if {id} is numeric.
});
```

Доступ к значению параметров роута

Если вам нужно получить значение параметра роута не в нем, а, где-то еще, например, в фильтре или контроллере, то вы можете использовать `Route::input()`:

```
Route::filter('foo', function()
{
    if (Route::input('id') == 1)
    {
        //
    }
});
```

Фильтры роутов

Фильтры - удобный механизм ограничения доступа к определённому роуту, что полезно при создании областей сайта только для авторизованных пользователей. В Laravel изначально включено несколько фильтров, в том числе `auth`, `auth.basic`, `guest` and `csrf`. Они определены в файле `app/filters.php`.

Регистрация фильтра маршрутов:

```
Route::filter('old', function()
{
    if (Input::get('age') < 200)
    {
        return Redirect::to('home');
    }
```

```
});
```

Если фильтр возвращает значение, оно используется как ответ фреймворка (response) на сам запрос (request) и обработчик маршрута не будет вызван, и все after-фильтры тоже будут пропущены.

Привязка фильтра к роуту:

```
Route::get('user', array('before' => 'old', function()
{
    return 'You are over 200 years old!';
}));
```

Сочетания фильтра и привязки роута к контроллеру

```
Route::get('user', array('before' => 'old', 'uses' => 'UserController@showProfile'));
```

Привязка нескольких фильтров к роуту

```
Route::get('user', array('before' => 'auth|old', function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

Привязка нескольких фильтров к роуту при помощи массива

```
Route::get('user', array('before' => array('auth', 'old'), function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

Передача параметров для фильтра

```
Route::filter('age', function($route, $request, $value)
{
    //
});

Route::get('user', array('before' => 'age:200', function()
{
    return 'Hello World';
}));
```

Фильтры типа after (выполняющиеся после запроса, если он не был отменён фильтром before - прим. пер.) получают $response как свой третий аргумент:

```
Route::filter('log', function($route, $request, $response)
{
    //
});
```

Фильтры по шаблону

Вы можете также указать, что фильтр применяется ко всем роутам, URI (путь) которых соответствует шаблону.

```
Route::filter('admin', function()
{
    //
});

Route::when('admin/*', 'admin');
```

В примере выше фильтр admin будет применён ко всем маршрутам, адрес которых начинается с admin/. Звёздочка (*) используется как символ подстановки и соответствует любому набору символов, в том числе пустой строке.

Вы также можете привязывать фильтры, зависящие от типа HTTP-запроса:

```
Route::when('admin/*', 'admin', array('post'));
```

Классы фильтров

Для продвинутой фильтрации вы можете использовать классы вместо функций-замыканий. Так как такие фильтры будут создаваться с помощью IoC-контейнера, то вы можете использовать внедрение зависимостей (Dependency

Injection) для лучшего тестирования.

Определение класса для фильтра:

```
Route::filter('foo', 'FooFilter');
```

По умолчанию будет вызван метод `filter` класса `FooFilter`:

```
class FooFilter {

    public function filter()
    {
        // Логика фильтра...
    }

}
```

Вы можете изменить это дефолтное поведение и указать метод явно:

```
Route::filter('foo', 'FooFilter@foo');
```

Именованные роуты

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

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

Также можно указать контроллер и его действие:

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

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

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

$redirect = Redirect::route('profile');
```

Получить имя текущего выполняемого маршрута 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(array('before' => '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.
    });
});
```

Внутри группы вы можете указать параметр `namespace` , чтобы не прописывать неймспейсы к каждому контроллеру:

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

Поддоменные роуты

Роуты Laravel способны работать и с поддоменами по их маске и передавать в ваш обработчик параметры из

шаблона.

Регистрация роута по поддомену:

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

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

});
```

Префикс пути

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

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

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

});
```

Привязка моделей к роутам

Привязка моделей - удобный способ передачи экземпляров моделей в ваш роут. Например, вместо передачи ID пользователя вы можете передать модель User, которая соответствует данному ID, целиком. Для начала используйте метод `Route::model` для указания модели, которая должна быть использована вместо данного параметра.

Привязка параметра к модели

```
Route::model('user', 'User');
```

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

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

Из-за того, что мы ранее привязали параметр `{user}` к модели `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 - прим. пер.).

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`:

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

Ошибки 404

Есть два способа вызвать исключение 404 (Not Found) из маршрута. Первый - методом `App::abort`:

```
App::abort(404);
```

Второй - бросив исключение класса или потомка класса `Symfony\Component\HttpKernel\Exception\NotFoundHttpException`.

Больше информации о том, как обрабатывать исключения 404 и отправлять собственный ответ на такой запрос содержится в разделе об ошибках.

Роуты в контроллер

Laravel позволяет вам регистрировать маршруты не только в виде функции-замыкания, но и классов-контроллеров и даже создавать контроллеры ресурсов.

Больше информации содержится в разделе о контроллерах.

Запросы и входные данные

Текущие входные данные

Вы можете получить доступ ко всем данным, переданным приложению, используя всего несколько простых методов. Вам не нужно думать о том, какой тип HTTP-запроса был использован (GET, POST и т.д.) - методы работают одинаково для любого из них.

Получение переменной

```
$name = Input::get('name');
```

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

```
$name = Input::get('name', 'Sally');
```

Была ли передана переменная?

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

Получение всех переменных запроса:

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

Получение некоторых переменных:

```
// Получить только перечисленные:
$input = Input::only('username', 'password');

// Получить все, кроме перечисленных:
$input = Input::except('credit_card');
```

> **Примечание:** Некоторые JavaScript-библиотеки, такие как Backbone, могут передавать переменные в виде JSON. Вне зависимости от этого `Input::get` будет работать одинаково.

Cookies

Все cookie, создаваемые Laravel, шифруются и подписываются специальным кодом - таким образом, если они изменятся на клиенте, то они станут неверными и не будут приниматься фреймворком.

Чтение cookie

```
$value = Cookie::get('name');
```

Добавление cookie к ответу (response)

```
$response = Response::make('Hello World');

$response->withCookie(Cookie::make('name', 'value', $minutes));
```

Помещение cookie в очередь на выставление

Может возникнуть ситуация, когда вам нужно установить определенную cookie в том месте кода, где response (ответ) еще не создан. В таком случай используйте `Cookie::queue()` - cookie будет автоматически добавлена в response после его окончательного формирования.

```
Cookie::queue($name, $value, $minutes);
```

Создание cookie, которая хранится вечно

```
$cookie = Cookie::forever('name', 'value');
```

Старые входные данные

Вам может пригодиться сохранение входных данных между двумя запросами. Например, после проверки формы на корректность вы можете заполнить её старыми значениями в случае ошибки.

Сохранение всего ввода для следующего запроса

```
Input::flash();
```

Сохранение некоторых переменных для следующего запроса

```
// Сохранить только перечисленные:
Input::flashOnly('username', 'email');

// Сохранить все, кроме перечисленных:
Input::flashExcept('password');
```

Обычно требуется сохранить входные данные при переадресации на другую страницу - это делается легко:

```
return Redirect::to('form')->withInput();

return Redirect::to('form')->withInput(Input::except('password'));
```

> **Примечание:** Вы можете сохранять и другие данные внутри сессии, используя класс Session.

Получение старых входных данных

```
Input::old('username');
```

Файлы

Получение объекта загруженного файла

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

Определение успешной загрузки файла

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

Метод `file` возвращает объект класса `Symfony\Component\HttpFoundation\File\UploadedFile`, который в свою очередь расширяет стандартный класс `SplFileInfo`, который предоставляет множество методов для работы с файлами.

Проверка загруженного файла на валидность

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

Перемещение загруженного файла

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

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

Получение пути к загруженному файлу

```
$path = Input::file('photo')->getRealPath();
```

Получение имени файла на клиентской системе (до загрузки)

```
$name = Input::file('photo')->getClientOriginalName();
```

Получение расширения загруженного файла

```
$extension = Input::file('photo')->getClientOriginalExtension();
```

Получение размера загруженного файла

```
$size = Input::file('photo')->getSize();
```

Определение MIME-типа загруженного файла

```
$mime = Input::file('photo')->getMimeType();
```

Информация о запросе (request)

Класс `Request` содержит множество методов для изучения входящего запроса в вашем приложении. Он расширяет класс `Symfony\Component\HttpFoundation\Request`. Ниже - несколько полезных примеров.

Получение URI (пути) запроса

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

Определение метода запроса (GET, POST и т.п.)

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

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

Соответствует ли запрос маске пути?

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

Получение URL запроса

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

Извлечение сегмента URI (пути)

```
$segment = Request::segment(1);
```

Чтение заголовка запроса

```
$value = Request::header('Content-Type');
```

Чтение значения из $_SERVER

```
$value = Request::server('PATH_INFO');
```

Используется ли HTTPS ?

```
if (Request::secure())
{
    //
}
```

Это ajax-запрос ?

```
if (Request::ajax())
{
    //
}
```

Запрос имеет формат JSON ?

```
if (Request::isJson())
{
    //
```

```
}
```

Ожидается, что ответ будет в формате JSON ?

```
if (Request::wantsJson())
{
    //
}
```

Определение ожидаемого формата ответа

Метод `Request::format` основывается на данных http-заголовка `Accept`

```
if (Request::format() == 'json')
{
    //
}
```

Ответ (response) и шаблоны (views)

Ответ: основы

Ответ в виде возврата строки из роута:

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

Создание собственного ответа

Объект `Response` наследует класс `Symfony\Component\HttpFoundation\Response` который предоставляет набор методов для построения HTTP-ответа.

```
$response = Response::make($contents, $statusCode);

$response->header('Content-Type', $value);

return $response;
```

Добавление cookie к ответу

```
$cookie = Cookie::make('name', 'value');

return Response::make($content)->withCookie($cookie);
```

Редиректы

Простой редирект

```
return Redirect::to('user/login');
```

Переадресация с одноразовыми переменными сессии:

```
return Redirect::to('user/login')->with('message', 'Войти не удалось');
```

> **Примечание:** Метод `with` сохраняет данные в сессии, поэтому вы можете прочитать их, используя обычный метод `Session::get`.

Переадресация на именованный роут

```
return Redirect::route('login');
```

Переадресация на именованный роут с параметрами

```
return Redirect::route('profile', array(1));
```

Переадресация на именованный роут с именованными параметрами

```
return Redirect::route('profile', array('user' => 1));
```

Переадресация на метод контроллера

```
return Redirect::action('HomeController@index');
```

Переадресация на метод контроллера с параметрами

```
return Redirect::action('UserController@profile', array(1));
```

Переадресация на метод контроллера с именованными параметрами

```
return Redirect::action('UserController@profile', array('user' => 1));
```

Вьюхи

Вьюхи (views, шаблон) обычно содержат HTML-код вашего приложения и представляют собой удобный способ разделения бизнес-логики и логики отображения информации. Вьюхи хранятся в папке `app/views`.

Простая вьюха может иметь такой вид:

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

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

Из этой вьюхи можно сформировать ответ в браузер, например, следующим образом:

```
Route::get('/', function()
{
    return View::make('greeting', array('name' => 'Тейлор'));
});
```

Второй параметр, переданный в `View::make` - массив данных, которые будут доступны внутри вьюхи.

Передача переменных в вьюху

```
// припомощи метода with()
$view = View::make('greeting')->with('name', 'Стив');

// при помощи "магического" метода
$view = View::make('greeting')->withName('Стив');
```

В примере выше переменная `$name` будет доступна в шаблоне и будет иметь значение `Стив`.

Можно передать массив данных в виде второго параметра для метода `make`:

```
$view = View::make('greetings', $data);
```

Вы также можете установить глобальную переменную, которая будет видна во всех шаблонах:

```
View::share('name', 'Стив');
```

Передача вложенного шаблона в шаблон

Иногда вам может быть нужно передать вьюху внутрь другой вьюхи. Например, передаем `app/views/child/view.php` внутрь `app/views/greeting.php`:

```
$view = View::make('greeting')->nest('child', 'child.view');

// с передачей переменных
$view = View::make('greeting')->nest('child', 'child.view', $data);
```

Затем вложенная вьюха может быть отображёна в `app/views/greeting.php`:

```
<html>
    <body>
        <h1>Привет!</h1>
        <?php echo $child; ?>
    </body>
</html>
```

> **Примечание** Для работы с вложенными вьюхами смотрите также `@include` шаблонизатора Blade

Композеры

Композеры (view composers) - функции-замыкания или методы класса, которые вызываются, когда вьюха рендерится в строку. Если у вас есть данные, которые вы хотите привязать к вьюхе при каждом её формировании, то композеры помогут вам выделить такую логику в отдельное место. Можно сказать, композеры - это модели вьюх.

Регистрация композера

```
View::composer('profile', function($view)
```

```
{
    $view->with('count', User::count());
});
```

Теперь при каждом отображении шаблона `profile` к нему будет привязана переменная `count`.

Вы можете привязать композер сразу к нескольким вьюхам:

```
View::composer(array('profile','dashboard'), function($view)
{
    $view->with('count', User::count());
});
```

Если вам больше нравится использовать классы (что позволит вам регистрировать несколько композеров в IoC Container и ресолвить нужный в сервис-провайдере), то вы можете сделать так:

```
View::composer('profile', 'ProfileComposer');
```

Класс композера должен иметь следующий вид:

```
class ProfileComposer {

    public function compose($view)
    {
        $view->with('count', User::count());
    }

}
```

Регистрация нескольких композеров

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

```
View::composers(array(
    'AdminComposer' => array('admin.index', 'admin.profile'),
    'UserComposer' => 'user',
));
```

> **Примечание** Обратите внимание, что нет строгого правила, где должны храниться классы-композеры. Вы можете поместить их в любое место, где их сможет найти автозагрузчик в соответствии с директивами в вашем файле `composer.json`.

Криейтор (view creators)

Криейторы вьюх работают почти так же, как композеры , но вызываются сразу после создания объекта вьюхи, а не во время её отображения в строку. Для регистрации используйте метод `creator`:

```
View::creator('profile', function($view)
{
    $view->with('count', User::count());
});
```

Особые ответы (response)

Создание JSON-ответа

```
return Response::json(array('name' => 'Стив', 'state' => 'CA'));
```

Создание JSONP-ответа

```
return Response::json(array('name' => 'Стив', 'state' => 'CA'))->setCallback(Input::get('callback'));
```

Создание ответа передачи файла

```
return Response::download($pathToFile);

return Response::download($pathToFile, $name, $headers);
```

> **Примечание** Symfony HttpFoundation, при помощи которого реализована отдача файла, требует, чтобы имя файла состояло из ASCII-символов.

Макросы ответов

Если вы хотите использовать особый ответ в нескольких роутах или контроллерах, то вы можете определить свой

макрос ответа:

```
Response::macro('caps', function($value)
{
    return Response::make(strtoupper($value));
});
```

Использование:

```
return Response::caps('foo');
```

Определения макросов ответов должны располагаться в одном из ваших старт-файлов.

Контроллеры

Простейшие контроллеры

Вместо того, чтобы писать код в роутах, вы можете организовать его, используя класс Controller. Контроллеры могут группировать связанную логику в отдельные классы, а кроме того использовать дополнительные возможности Laravel, такие как автоматическое внедрение зависимостей.

Контроллеры обычно хранятся в папке `app/controllers`, а этот путь по умолчанию зарегистрирован в настройке `classmap` вашего файла `composer.json`. Но вообще-то контроллеры могут находиться в любой папке или подпапке проекта - лишь бы была настроена их загрузка в Composer.

Вот пример простейшего класса контроллера:

```
class UserController extends BaseController {

    /**
     * Отобразить профиль соответствующего пользователя.
     */
    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

Все контроллеры должны наследовать класс `BaseController`. Этот класс также может хранится в папке `app/controllers` и в него можно поместить общую логику для других контроллеров. The `BaseController` расширяет стандартный класс Laravel, `Controller` class. Теперь, определив контроллер, мы можем зарегистрировать роут для его действия (action):

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

Если вы решили организовать ваши контроллеры в пространстве имён, просто используйте полное имя класса при определении маршрута:

```
Route::get('foo', 'Namespace\FooController@method');
```

> **Примечание:** Так как Laravel использует Composer для автоматической загрузки php-классов, контроллеры могут располагаться в любом месте фоеймворка, лишь бы Composer знал, как их загрузить. Не обязательно держать контролеры в папке `app/controllers`. Роутинг к контроллерам полностью отвязан от особенностей расположения файлов фреймворка.

Вы также можете присвоить имя этому роуту:

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

Вы можете получить URL к экшну (действию, методу контроллера) методом `URL::action`:

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

Получить имя экшна, которое выполняется в данном запросе, можно методом `currentRouteAction`:

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

Фильтры контроллеров

Фильтры могут указываться для роутов в контроллеры аналогично тому, как они указываются для обычных роутов:

```
Route::get('profile', array('before' => 'auth',
            'uses' => 'UserController@showProfile'));
```

Однако вы можете указывать их и внутри самого контроллера:

```
class UserController extends BaseController {

    /**
     * Создать экземпляр класса UserController.
     */
    public function __construct()
    {
        $this->beforeFilter('auth');

        $this->beforeFilter('csrf', array('on' => 'post'));

        $this->afterFilter('log', array('only' =>
                            array('fooAction', 'barAction')));
    }

}
```

Можно устанавливать фильтры в виде функции-замыкания:

```
class UserController extends BaseController {

    /**
     * Создать экземпляр класса UserController.
     */
    public function __construct()
    {
        $this->beforeFilter(function()
        {
            //
        });
    }

}
```

Можно выносить фильтр в метод контроллера:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('@filterRequests');
    }

    /**
     * Filter the incoming requests.
     */
    protected function filterRequests($route, $request)
    {
        //
    }

}
```

RESTful-контроллеры

Laravel позволяет вам легко создавать единый маршрут для обработки всех действий контроллера используя простую схему именования REST. Для начала зарегистрируйте маршрут методом `Route::controller`:

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

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

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
```

```
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

В этом случае фреймворк будет обрабатывать обращение (GET-запрос) к /users и POST-запрос на /users/profile. На остальные запросы будет отдаваться 404. Метод `index` обрабатывает корневой URI контроллера.

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

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

Имена роутов

Имена роутов для RESTful-контроллеров можно задавать следующим образом:

```
Route::controller('manager', 'ManagerController', array(
    'getIndex' => 'manager.index',
    'getLogout' => 'manager.logout'
));
```

Контроллеры ресурсов

Контроллеры ресурсов упрощают построение RESTful-контроллеров, работающих с ресурсами. Например, вы можете создать контроллер, обрабатывающий фотографии, хранимые вашим приложением. Вы можете быстро создать такой контроллер с помощью команды `controller:make` интерфейса (Artisan) и метода `Route::resource`.

Для создания контроллера выполните следующую консольную команду:

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

Теперь мы можем зарегистрировать его как контроллер ресурса:

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

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

Запросы, обрабатываемые контроллером ресурсов:

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

Иногда вам может быть нужно обрабатывать только часть всех возможных действий:

```
php artisan controller:make PhotoController --only=index,show

php artisan controller:make PhotoController --except=index
```

Вы можете указать этот набор и при регистрации маршрута:

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

// либо:

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

По умолчанию имена у роутов совпадают с названиями экшнов контроллеров ресурсов. Однако, вы можете изменить это:

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

Обработка неопределённых методов

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

Определение Catch-All метода

```
public function missingMethod($parameters = array())
{
    //
}
```

Ошибки и логирование

Конфигурация

Логгер регистрируется в старт-файле `app/start/global.php`. По умолчанию логи пишутся в один файл, но вы можете изменить это поведение. Для ведения логов Laravel использует модуль Monolog.

Например, если вы не хотите, чтобы все логи писались в один огромный файл, а хотите разбивать свои лог-файлы по дням, то вам нужно вписать в `app/start/global.php` следующее:

```
$logFile = 'laravel.log';
Log::useDailyFiles(storage_path().'/logs/'.$logFile);
```

Детализация ошибок

По умолчанию в Laravel включена детализация ошибок, происходящих в вашем приложении. Это значит, что при их возникновении будет отображена страница с цепочкой вызовов и текстом ошибки. Вы можете отключить детализацию ошибок установкой настройки debug файла `app/config/app.php` в значение `false`.

> **Примечание** Настоятельно рекомендуется отключать детализацю ошибок для рабочей среды `production`

Обработка ошибок

По умолчанию, файл `app/start/global.php` содержит обработчик всех исключений:

```
App::error(function(Exception $exception)
{
    Log::error($exception);
});
```

Это самый примитивный обработчик. Однако вы можете зарегистрировать несколько обработчиков, если вам это нужно. Они будут вызываться в зависимости от типа `Exception`, указанного в их первом аргументе. Например, вы можете создать обработчик только для ошибок `RuntimeException`:

```
App::error(function(RuntimeException $exception)
{
    // Обработка исключения...
});
```

Если обработчик возвращает ответ, он будет отправлен в браузер и никакие другие обработчики вызваны не будут:

```
App::error(function(InvalidUserException $exception)
{
    Log::error($exception);

    return 'Извини! Что-то не так с этим аккаунтом!';
});
```

Вы можете зарегистрировать обработчик критических ошибок PHP методом `App::fatal`:

```
App::fatal(function($exception)
{
    //
});
```

Если у вас несколько обработчиков ошибок в приложении, их определения должны быть расположены в порядке от наиболее общего до наиболее конкретного. Наример, обработчик, который обрабатывает `Exception` должен быть определен раньше чем тот, который обрабатывает что-то вроде `Illuminate\Encryption\DecryptException`

Где располагать обрабтчики ошибок

Нет общепринятого места для определения обработчиков ошибок - вы вольны располагать их где вам удобно. Одним из вариантов может быть файл `start/global.php`. Если вы не хотите перегружать этот файл, то можете создать,

например, app/errors.php и подключать его в start/global.php. Третий вариант - создать сервис-провайдер, в котором вы будете регистрировать обработчики ошибок. Нет однозначно правильного ответа, где это делать - выберите то, что будет удобно лично для вас.

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:

```
App::abort(404);
```

Опционально, вы можете установить свой ответ для возврата в браузер:

```
App::abort(403, 'У вас нет прав для просмотра этой страницы.');
```

Эти исключения могут быть возбуждены на любом этапе обработки запроса.

Обработка 404

Вы можете зарегистрировать обработчик для всех ошибок 404 ("Не найдено") в вашем приложении, что позволит вам отображать собственную страницу 404:

```
App::missing(function($exception)
{
    return Response::view('errors.missing', array(), 404);
});
```

Логирование

Стандартный механизм логирования представляет собой простую надстройку над мощной системой Monolog. По умолчанию Laravel настроен так, чтобы хранить все логи в одном файле - app/storage/logs/laravel.log. Вы можете записывать в него 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 (в порядке возрастания - прим. пер.): **debug**, **info**, **notice**, **warning**, **error**, **critical** и **alert**.

В метод записи можно передать массив данных о текущем состоянии:

```
Log::info('Log message', array('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)
{
    //
});
```

Безопасность

Конфигурация

Laravel стремится сделать реализацию авторизации максимально простой. Фактически, после установки фреймворка почти всё уже настроено. Настройки хранятся в файле `app/config/auth.php`, который содержит несколько хорошо документированных параметров для настройки поведения методов аутентификации.

"Из коробки" приложение Laravel включает в себя модель `User` в папке `app/models`, которая может использоваться вместе с дефолтным драйвером аутентификации Eloquent. При создании таблицы для данной модели убедитесь, что поле пароля принимает как минимум 60 символов.

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

> **Примечание:** Перед тем как начать, пожалуйста, убедитесь, что таблица `users` (или другая, в которой хранятся пользователи) содержит nullable (с возможностью содержать NULL) столбец `remember_token` длиною 100 символов (VARCHAR(100)). Этот столбец испольдуется для хранения токена, когда пользователь при логине ставит галку "запомнить меня".

Хранение паролей

Класс `Hash` содержит методы для безопасного хэширования с помощью Bcrypt.

Хэширование пароля по алгоритму Bcrypt:

```
$password = Hash::make('secret');
```

Проверка пароля по хэшу:

```
if (Hash::check('secret', $hashedPassword))
{
    // Пароль подходит
}
```

Проверка на необходимость перехэширования пароля:

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

Аутентификация пользователей

Для аутентификации пользователя в вашем приложении вы можете использовать метод `Auth::attempt`.

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{
    return Redirect::intended('dashboard');
}
```

Заметьте, что поле `email` не обязательно и оно используется только для примера. Вы должны использовать любое поле, которое соответствует имени пользователя в вашей БД. Метод `Redirect::intended` отправит пользователя на URL, который он пытался просмотреть до того, как запрос был перехвачен фильтром аутентификации. Также в этом методе можно задать дополнительный URL, куда будет осуществлен переход, если первый URL не доступен.

Когда вызывается метод `attempt`, запускается событие `auth.attempt`. При успешной авторизации также запускается событие `auth.login`.

Проверка авторизации пользователя

Для определения того, авторизован ли пользователь или нет, можно использовать метод check.

```
if (Auth::check())
{
    // Пользователь уже вошёл в систему
}
```

Если вы хотите предоставить функциональность типа "запомнить меня", то вы можете передать `true` вторым параметром к методу `attempt`, который будет поддерживать авторизацию пользователя без ограничения по времени (пока он вручную не выйдет из системы). В такоем случае у вашей таблицы `users` должен быть строковой столбец `remember_token` для хранения токена пользователя.

```
if (Auth::attempt(array('email' => $email, 'password' => $password), true))
{
    // Пользователь был запомнен
}
```

Примечание: если метод `attempt` вернул `true`, то пользователь успешно вошёл в систему.

Имеет ли пользователь токен "запомнить меня"

Метод `viaRemember` позволяет узнать, вошел ли пользователь при помощи фичи "запомнить меня".

```
if (Auth::viaRemember())
{
    // Пользователь вошел, так как ранее ставил галку "запомнить меня"
}
```

Авторизация пользователя с использованием условий

Вы также можете передать дополнительные условия для запроса к таблице:

```
if (Auth::attempt(array('email' => $email, 'password' => $password, 'active' => 1)))
{
    // Вход, если пользователь активен, не отключен и существует.
}
```

> **Примечание** Для повышения безопасности после аутентификации фреймворк регенерирует ID сессии пользователя.

Доступ к залогиненному пользователю

Как только пользователь авторизован вы можете обращаться к модели `User` и её свойствам.

```
$email = Auth::user()->email;
```

Для простой аутентификации пользователя по ID используется метод `loginUsingId`:

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

Проверка данных для входа без авторизации

Метод `validate` позволяет вам проверить данные для входа без осуществления самого входа.

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

Аутентификация пользователя на один запрос

Вы также можете использовать метод once для авторизации пользователя в системе только для одного запроса. Сессии и cookies не будут использованы.

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

Выход пользователя из приложения

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

Ручная авторизация

Если вам нужно войти существующим пользователем, просто передайте его модель в метод `login`:

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

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

Это эквивалентно аутентификации пользователя через его данные методом `attempt`.

Аутентификация и роутинг

Вы можете использовать Фильтры роутов, чтобы позволишь только залогиненным пользователям обращаться к заданному роуту или группе роутов. Изначально Laravel содержит фильтр `auth`, который содержится в файле `app/filters.php`.

Защита роута аутентификацией

```
Route::get('profile', array('before' => 'auth', function()
{
    // Доступно только залогиненным пользователям.
}));
```

Защита от подделки запросов (CSRF)

Laravel предоставляет простой способ защиты вашего приложения от подделки межсайтовых запросов (cross-site request forgeries, CSRF).

Вставка CSRF-ключа в форму

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

Проверка переданного CSRF-ключа

```
Route::post('register', array('before' => 'csrf', function()
{
    return 'Вы передали верный ключ!';
}));
```

HTTP-аутентификация

HTTP Basic Authentication - простой и быстрый способ аутентификации пользователей вашего приложения без создания дополнительной страницы входа. При HTTP-аутентификации форму для ввода логина и пароля показывает браузер, в виде всплывающего окна. Для HTTP-аутентификации используйте фильтр `auth.basic`:

Защита роута фильтром HTTP-аутентификации

```
Route::get('profile', array('before' => 'auth.basic', function()
{
    // Доступно только залогиненным пользователям.
}));
```

По умолчанию, фильтр `basic` будет использовать поле `email` модели объекта при аутентификации. Если вы хотите использовать иное поле, можно передать его имя первым параметром методу `basic` в файле `app/filters.php`:

```
Route::filter('auth.basic', function()
{
    return Auth::basic('username');
});
```

Авторизация без запоминания состояния

Вы можете использовать HTTP-авторизацию без установки cookies в сессии, что особенно удобно для аутентификации в API. Для этого зарегистрируйте фильтр, вовращающий результат вызова `onceBasic`.

```
Route::filter('basic.once', function()
{
    return Auth::onceBasic();
```

```
});
```

Если вы используете Apache + PHP FastCGI, HTTP-аутентификация не будет работать "из коробки". Вам нужно добавить следующие строки в свой `.htaccess`:

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

Сброс забытого пароля

Модель и таблица

Восстановление забытого пароля - очень распространенная вещь в веб-приложениях. Чтобы не заставлять вас писать его вновь и вновь, Laravel предоставляет встроенные удобные методы для совершения подобных операций. Для начала убедитесь, что ваша модель `User` реализует (implements) интерфейс `Illuminate\Auth\Reminders\RemindableInterface`. Модель `User`, которая идет с фреймворком, уже реализует его.

Реализация RemindableInterface

```
class User extends Eloquent implements RemindableInterface {

    public function getReminderEmail()
    {
        return $this->email;
    }
}
```

Создание таблицы токенов сброса пароля

Далее, должна быть создана таблица для хранения токенов запросов сброса пароля. Для создания такой таблицы существует artisan-команда `auth:reminders-table`.

```
php artisan auth:reminders-table

php artisan migrate
```

Контроллер восстановления пароля

Чтобы автоматически создать контроллер восстановления пароля, воспользуйтесь командой `auth:reminders-controller`. В папке `controllers` будет создан `RemindersController.php`:

```
php artisan auth:reminders-controller
```

Созданный контроллер содержит метод `getRemind`, который показывает форму для напоминания пароля. Вам надо создать эту форму во вьюхе `password.remind` (файл remind.blade.php в папке views/password - см. view). Форма должна отправлять POST с email на метод `postRemind`

Простейший пример `password.remind`:

```
<form action="{{ action('RemindersController@postRemind') }}" method="POST">
    <input type="email" name="email">
    <input type="submit" value="Send Reminder">
</form>
```

Метод `postRemind` уже есть в сгенеренном `RemindersController.php`. Приняв `email` POST-запросом, контроллер отсылает на этот адрес письмо с подтверждением. Если оно отсылается нормально, в сессию во flash-переменную `status` заносится сообщение об успешной отправке. Если нет - во flash-переменную `error` заносится текст ошибки.

Для модификации собщения, которое уйдет пользователю на почту, вы можете изменить в контроллере вызов `Password::remind` на, например, такое:

```
Password::remind(Input::only('email'), function($message)
{
    $message->subject('Password Reminder');
});
```

Пользователь получит письмо со ссылкой на метод `getReset`, с токеном для индентификации пользователя. Этот метод вызывает вьюху `password.reset` (файл `reset.blade.php` в папке `views/password`), в которой должна быть форма для смены пароля, со скрытым полем `token` и полями `email`, `password`, and `password_confirmation`, например такая:

```
<form action="{{ action('RemindersController@postReset') }}" method="POST">
```

```
    <input type="hidden" name="token" value="{{ $token }}">
    <input type="email" name="email">
    <input type="password" name="password">
    <input type="password" name="password_confirmation">
    <input type="submit" value="Reset Password">
</form>
```

Метод `postReset` производит замену паролей в вашем сторадже. По умолчанию считается, что пользователи хранятся в БД, с которой умеет работать Eloquent - ожидается $user, который передается в функцию-замыкание имеет метод `save`. Если это не так, измените функцию-замыкание в аргументе `Password::reset` в контроллере `RemindersController` исходя из вашей архитектуры приложения.

Если смена пароля прошла удачно, пользователь редиректится на главную страницу вашего приложения (вы можете изменить URL редиректа, если хотите). Если нет - пользователь редиректится обратно с установкой flash-переменной `error`.

Валидация паролей

По умолчанию `Password::reset` валидирует пароль пользователя исходя из двух правил - введенные пароли должны совпадать и пароль должен быть больше или равен 6 символам. Если вы хотите расширить валидацию паролей, вы можете определить свой `Password::validator`:

```
Password::validator(function($credentials)
{
    return strlen($credentials['password']) >= 6;
});
```

> **Примечание** Токены сброса пароля валидны в течении одного часа. Вы можете изменить это в `app/config/auth.php`, параметр `reminder.expire`.

Шифрование

Laravel предоставляет функции устойчивого шифрования по алгоритму AES с помощью расширения `mcrypt` для PHP.

Шифрование строки

```
$encrypted = Crypt::encrypt('секрет');
```

> **Примечание:** обязательно установите 16, 24 или 32-значный ключ `key` в файле `app/config/app.php`. Если этого не сделать, зашифрованные строки не будут достаточно надежными.

Расшифровка строки

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Изменение режима и алгоритма шифрования

Вы также можете установить свой алгоритм и режим шифрования:

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

Драйвера аутентификации

Laravel из коробки предлагает для аутентификации драйвера `database` и `eloquent`. Чтобы узнать больше о том, как добавлять свои драйвера, изучите документацию по расширению системы аутентификации.

Кэш

Настройка

Laravel предоставляет унифицированное API для различных систем кэширования. Настройки кэша содержатся в файле `app/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 упакованные объекты кэша в файловой системе. Для больших приложений рекомендуется использование систем кэширования в памяти - таких как `Memcached` или `APC`.

Использование кэша

Запись нового элемента в кэш

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

Использование объекта 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 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 быть нужно 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.

Удаление элемента из кэша

```
Cache::forget('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 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` - чтобы старые записи чистились в автоматическом режиме.

Обращение к группе элементов кэша

При помощи тэгов вы можете объединять элементы кэша в группы, а затем сбрасывать всю группу целиком. Для доступа к группе используйте метод `tags`.

Элемент кэша можно в ключать в одну или несколько групп-тэгов. Список тэгов-групп можно перечислять через запятую в аргументах функции, или как элементы массива:

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

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

Вместе с объединения с тэгами Вы можете использовать обычные операции над элементами кэша, такие как чтение, запись, `remember`, `forever`, `rememberForever`, `increment`, `decrement`.

Получение элементов из тэгированного кэша

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

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

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

Тэги кэша нужны главным образом для того, чтобы можно было легко удалить несколько элементов кэша. Например, это выражение удалит все элементы кэша, которые содержатся в группах `people`, `authors` или в обоих сразу:

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

И "Anne" и "John" будут удалены из кэша. Для сравнения, это выражение удалит только "John":

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

Кэширование в базе данных

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

```
Schema::create('cache', function($table)
```

```
{
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

Расширение фреймворка

Введение

Laravel предоставляет вам множество точек для настройки поведения различных частей ядра библиотеки или даже полной замены. К примеру, хэширующие функции определены интерфейсом `HasherInterface`, который вы можете реализовать (implement) в зависимости от требований вашего приложения. Вы также можете расширить объект `Request`, добавив собственные удобные вспомогательные методы (helpers). Вы даже можете добавить новый драйвер авторизации, кэширования и сессии!

Расширение компонентов Laravel происходит двумя основными способами: привязка новой реализации через контейнер IoC или регистрация расширения через класс `Manager`, который реализует шаблон проектирования "Factory" ("Фабрика"). В этом разделе мы изучим различные методы расширения фреймворка и код, который для этого необходим.

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.

Где писать код ?

Данная документация описывает как расширять Laravel. Где именно писать этот код - зависит от вас. Например, для расширения `Cache` или `Auth` неплохим выбором будут start-файлы фреймворка. Те же расширения, которые должны подключаться максимально рано, например `Session`, должны располагаться в методе `register`сервис-провайдера вашего приложения.

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)
{
    // Вернуть объект типа Illuminate\Cache\Repository...
});
```

Первый параметр, передаваемый методу `extend` - имя драйвера. Это имя соответствует значению параметра `driver` файла настроек `app/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.

Для создания стороннего драйвера для кэша мы начнём с реализации интерфейса `Illuminate\Cache\StoreInterface`. Итак, наша реализация MongoDB будет выглядеть примерно так:

```
class MongoStore implements Illuminate\Cache\StoreInterface {
```

```
    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:

```
use Illuminate\Cache\Repository;

Cache::extend('mongo', function($app)
{
    return new Repository(new MongoStore);
});
```

Как вы видите в примере выше, можно использовать базовый класс `Illuminate\Cache\Repository` при создании нового драйвера для кэша. Обычно не требуется создавать собственный класс хранилища.

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` в основной папке вашего приложения. Например, если оно называется Snappy, вы можете поместить драйвер кэша в `app/Snappy/Extensions/MongoStore.php`. Впрочем, так как Laravel не имеет жёсткой структуры папок, вы можете организовать свои файлы, как вам удобно.

> **Примечание:** если у вас стоит вопрос о том, где должен располагаться определённый код, в первую очередь вспомните о сервис-провайдерах.

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

Расширение системы сессий должно быть зарегистрировано раньше, чем расширения других систем. Эта система стартует очень рано, до запуска старт-файлов. Поэтому вы должны регистрировать расширение этой системы в методе `register` сервис-провайдера вашего приложения, и ваш сервис-провайдер должен находиться **ниже** дефолтного `Illuminate\Session\SessionServiceProvider` в массиве `providers` файла `app\config\app.php`.

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 app/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\Auth\UserProviderInterface
});
```

Реализация UserProviderInterface ответственна только за то, чтобы получать нужный объект UserInterface 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 определение UserProviderInterface:

```
interface UserProviderInterface {

    public function retrieveById($identifier);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(UserInterface $user, array $credentials);

}
```

Метод retrieveById бычно получает числовой ключ, идентифицирующий пользователя - такой, как первичный ключ в MySQL. Метод должен возвращать объект UserInterface, соответствующий переданному ID.

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 UserProviderInterfaceдавайте посмотрим на интерфейс UserInterface. Как вы помните, поставщик должен вернуть реализацию этого интерфейса из своих методов retrieveById и retrieveByCredentials.

```
interface UserInterface {

    public function getAuthIdentifier();
    public function getAuthPassword();
```

```
}
```

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/models` который реализует этот интерфейс, поэтому мы можете обратиться к этому классу, чтобы увидеть пример реализации.

Наконец, как только мы написали класс-реализацию `UserProviderInterface`, 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 `app/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 `app/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 функциональность 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:

```
class SnappyHashProvider extends Illuminate\Hashing\HashServiceProvider {

    public function boot()
    {
        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 `app/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.

Расширение Request

Класс `Request` это основополагающая часть фреймворка и создаётся в самом начале обработки запроса - gj'njve его расширение немного отличается от описанного выше.

Для начала расширьте класс, как это обычно делается:

```
<?php namespace QuickBill\Extensions;

class Request extends \Illuminate\Http\Request {

    // Здесь ваши собственные полезные методы

}
```

Как только класс расширен, откройте файл `bootstrap/start.php`. Это один из старт-файлов, подключаемых в самом

начале обработки запроса в вашем приложении. Заметьте, что самое первое, что здесь происходит - создание объекта $app:

```
$app = new \Illuminate\Foundation\Application;
```

Когда объект приложения создан, он создаст новый объект `Illuminate\Http\Request` и привяжет его к IoC-контейнеру с ключом `request`. Итак, нам нужен способ для указания нашего собственного класса, который должен использоваться в виде объекта запроса по умолчанию, верно? К счастью, метод объекта приложения `requestClass` выполняет как раз эту задачу! Итак, мы можем добавить эти строки в начало вашего файла `bootstrap/start.php`:

```
use Illuminate\Foundation\Application;

Application::requestClass('QuickBill\Extensions\Request');
```

Как только вы указали сторонний класс запроса, Laravel будет использовать его каждый раз при создании объекта `Request`, позволяя вам всегда под рукой собственную реализацию, даже в юнит-тестах.

События

Основы использования

Класс Event содержит простую реализацию концепции Observer ("Наблюдатель"), что позволяет вам подписываться на уведомления (listen for events) о событиях в вашем приложении.

Подписка на событие

```
Event::listen('auth.login', function($user)
{
    $user->last_login = new DateTime;

    $user->save();
});
```

Запуск события

```
$event = Event::fire('auth.login', array($user));
```

Подписка на событие с приоритетом

При подписывании на событие вы можете указать приоритет. Обработчики с более высоким приоритетом будут вызваны перед теми, чей приоритет ниже, а обработчики с одинаковым приоритетом будут вызываться в порядке их регистрации.

```
Event::listen('auth.login', 'LoginHandler', 10);

Event::listen('auth.login', 'OtherHandler', 5);
```

Прерывание обработки события

Иногда вам может быть нужно пропустить вызовы других обработчиков события. Вы можете сделать это, вернув значение false:

```
Event::listen('auth.login', function($event)
{
    // Обработка события

    return false;
});
```

Где писать код ?

Регистрировать обработчики событий можно практически везде - например, в старт-файле app/start/global.php. Или, если же этот файл у вас уже слишком сильно разросся, вы можете, например, сделать файл app/events.php и подключить (include) его в app/start/global.php. Если же вы предпочитаете ООП-подход, вы можете зарегистрировать ваши обработчики событий в одном из ваших сервис-провайдеров.

Обработчики по шаблону (wildcard)

Регистрация обработчика по шаблону

При регистрации обработчика вы можете использовать звёздочки (*) для привязки его ко всем подходящим событиям.

```
Event::listen('foo.*', function($param)
{
    // Обработка событий
});
```

Этот обработчик будет вызываться при любом событии, начинающемся foo..

Для определения того, какое именно событие из подходящих под foo.* поймано, используйте Event::firing()

```
Event::listen('foo.*', function($param)
{
    if (Event::firing() == 'foo.bar')
    {
        //
    }
});
```

Классы-обработчики

В некоторых случаях вы можете захотеть обрабатывать события внутри класса, а не функции-замыкания. Классы-обработчики берутся из IoC-контейнера, поэтому вы можете использовать все его возможности по автоматическому внедрению зависимостей.

Регистрация обработчика в классе

```
Event::listen('auth.login', 'LoginHandler');
```

Создание обработчика внутри класса

По умолчанию будет вызываться метод `handle` класса `LoginHandler`.

```
class LoginHandler {

    public function handle($data)
    {
        //
    }

}
```

Регистрация обработчика в другом методе

Если вы не хотите использовать метод `handle`, вы можете указать другое имя при регистрации.

```
Event::listen('auth.login', 'LoginHandler@onLogin');
```

Запланированные события

Регистрация цепочки событий

С помощью методов `queue` и `flush` вы можете запланировать события к запуску, но не запускать их сразу.

```
Event::queue('foo', array($user));
```

Регистрация "пускателя" событий

```
Event::flusher('foo', function($user)
{
    //
});
```

Наконец, вы можете запустить все запланированные события методом `flush`:

```
Event::flush('foo');
```

Классы-подписчики

Определение класса-подписчика

Классы-подписчики содержат обработчики множества событий. Подписчики должны содержать метод `subscribe`, которому будет передан экземпляр обработчика событий для регистрации.

```
class UserEventHandler {

    /**
     * Обработка событий входа пользователя в систему.
     */
    public function onUserLogin($event)
    {
        //
```

```
    }

    /**
     * Обработка событий выхода из системы.
     */
    public function onUserLogout($event)
    {
        //
    }

    /**
     * Регистрация всех обработчиков данного подписчика.
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen('auth.login', 'UserEventHandler@onUserLogin');

        $events->listen('user.logout', 'UserEventHandler@onUserLogout');
    }

}
```

Регистрация класса-подписчика

Как только класс-подписчик определён, он может быть зарегистрирован внутри класса `Event`.

```
$subscriber = new UserEventHandler;

Event::subscribe($subscriber);
```

Фасады

Введение

Фасады предоставляют "статический" интерфейс (`Foo::bar()`) к классам, доступным в IoC-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 IoC-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 фасаде на полученный объект.

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

Когда вы обращаетесь к любому статическому методу фасада Cache, Laravel получает объект cache из IoC 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');
```

Создание фасадов

Создать фасад в вашем приложении или пакете (package) довольно просто. Вам нужны только три вещи:

1. Биндинг (привязка) в IoC.

2. Класс-фасад.
3. Настройка для псевдонима фасада.

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

```
namespace PaymentGateway;

class Payment {

    public function process()
    {
        //
    }

}
```

Этот класс может находиться в `app/models` или в любой другой директории, где у composer настроена автозагрузка классов.

Нам нужно, чтобы этот класс извлекался из контейнера IoC, 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 поставщика в файле `app/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 алиас (alias, псевдоним) для этого фасада в массив `aliases` файла настроек `app/config/app.php`- тогда мы сможем вызывать метод `process` на классе `Payment`.

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

Об автозагрузке алиасов

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 моделях, которые содержат подсказки типов на неизвестные (неопределённые) классы. Единственное решение - не использовать алиасы 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
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\Reminders\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	

Формы и HTML

Открытие формы

Открытие формы

```
{{ Form::open(array('url' => 'foo/bar')) }}
    //
{{ Form::close() }}
```

По умолчанию используется метод POST, но вы можете указать другой метод:

```
echo Form::open(array('url' => 'foo/bar', 'method' => 'put'))
```

> **Примечание:** так как HTML-формы поддерживают только методы POST и GET, методы PUT и DELETE будут автоматически сэмулированы и переданы в скрытом поле `_method`.

Также вы можете открыть форму, которая может указывать на именованный(-е) маршрут или действие контроллера:

```
echo Form::open(array('route' => 'route.name'))

echo Form::open(array('action' => 'Controller@method'))
```

Вы можете передавать им параметры таким образом:

```
echo Form::open(array('route' => array('route.name', $user->id)))

echo Form::open(array('action' => array('Controller@method', $user->id)))
```

Если ваша форма будет загружать файлы, добавьте опцию `files`:

```
echo Form::open(array('url' => 'foo/bar', 'files' => true))
```

Защита от CRSF

Добавление CRSF-токена в форму

Laravel предоставляет простую защиту от подделки межсайтовых запросов. Сперва случайная последовательность символов (CSRF токен) помещается в сессию. Не переживайте - это делается автоматически. Эта строка также автоматически будет добавлена в вашу форму в виде скрытого поля. Тем не менее, если вы хотите сгенерировать HTML-код для этого поля, вы можете использовать метод `token`.

```
echo Form::token();
```

Присоединение CRSF-фильтра к маршруту

```
Route::post('profile', array('before' => 'csrf', function()
{
    //
}));
```

Привязка модели к форме

Открытие формы для модели

Зачастую вам надо представить содержимое модели в виде формы. Чтобы сделать это, используйте метод `Form::model`.

```
echo Form::model($user, array('route' => array('user.update', $user->id)))
```

Теперь, когда вы генерируете элемент формы - такой, как текстовое поле - значение свойства модели, соответствующее этому полю, будет присвоено ему автоматически. Так, для примера, значение текстового поля, названного `email`, будет установлено в значение свойства модели пользователя `email`. Но это еще не всё! Если в сессии будет переменная, чьё имя соответствует имени текстового поля, то будет использовано это значение, а не свойство модели.

Итак, приоритет выглядит следующим образом: 1. Flash-переменная сессии (старый ввод) 2. Напрямую переданные значения в запрос 3. Свойство модели

Это позволяет вам быстро строить формы, которые не только привязаны к свойствам модели, но и легко заполняются повторно, если произошла какая-нибудь ошибка на сервере.

> **Примечание:** при использовании `Form::model` всегда закрывайте форму при помощи метода `Form::close`!

Метки

Генерация элемента метки (label)

```
echo Form::label('email', 'Адрес e-mail');
```

Передача дополнительных атрибутов для тега

```
echo Form::label('email', 'Адрес e-mail', array('class' => 'awesome'));
```

> **Примечание:** после создания метки, любой элемент формы созданный вами, имя которого соответствует имени метки, автоматически получит её ID.

Текстовые и скрытые поля

Создание текстового поля ввода

```
echo Form::text('username');
```

Указание значения по умолчанию

```
echo Form::text('email', 'example@gmail.com');
```

> **Примечание:** методы *hidden* и *textarea* принимают те же параметры, что и метод *text*.

Генерация поля ввода пароля

```
echo Form::password('password');
```

Генерация других полей

```
echo Form::email($name, $value = null, $attributes = array());

echo Form::file($name, $attributes = array());
```

Чекбоксы и кнопки переключения

Генерация чекбокса или кнопки переключения (radio button)

```
echo Form::checkbox('name', 'value');

echo Form::radio('name', 'value');
```

Генерация нажатого чекбокса или радиокнопки, выбранной по умолчанию

```
echo Form::checkbox('name', 'value', true);

echo Form::radio('name', 'value', true);
```

Загрузка файлов

Генерация поля загрузки файла

```
echo Form::file('image');
```

> **Примечание:** Для загрузки файлов необходимо, чтобы форма бфла открыта с параметром `'files'=>true`.

Выпадающие списки

Генерация выпадающего списка

```
echo Form::select('size', array('L' => 'Большой', 'S' => 'Маленький'));
```

Генерация списка со значением по умолчанию

```
echo Form::select('size', array('L' => 'Большой', 'S' => 'Маленький'), 'S');
```

Генерация списка с группами (optgroup)

```
echo Form::select('animal', array(
    'Кошки' => array('leopard' => 'Леопард'),
    'Собаки' => array('spaniel' => 'Спаниель'),
));
```

Кнопки

Генерация кнопки отправки формы

```
echo Form::submit('Нажми меня!');
```

> **Примечание:** вам нужно создать кнопку (<button>)? Используйте метод *button* - он принимает те же параметры, что *submit*.

Макросы

Регистрация макроса для Form

К классу Form можно легко добавлять собственные методы; они называются "макросами".

Вот как это работает. Сперва зарегистрируйте макрос с нужным именем и функцией-замыканием.

```
Form::macro('myField', function()
{
    return '<input type="awesome">';
});
```

Теперь вы можете вызвать макрос, используя его имя:

Вызов макроса

```
echo Form::myField();
```

Формирование URL

Урлы можно не только задавать явным образом, но и формировать исходя из имени роута назначения или названия экшна контроллера - см. helpers.

Хелперы

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');
```

Примечание:

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 `app/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');
```

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();
```

Inversion of Control (обратное управление, IoC)

Введение

Класс-контейнер обратного управления IoC (Inversion of Control) Laravel - мощное средство для управления зависимостями классов. Внедрение зависимостей - это способ исключения вшитых (hardcoded) взаимосвязей классов. Вместо этого зависимости определяются во время выполнения, что даёт бОльшую гибкость благодаря тому, что они могут быть легко изменены.

Понимание IoC-контейнера Laravel необходимо для построения больших приложений, а также для внесения изменений в код ядра самого фреймворка.

Основы использования

Помещение в контейнер

Есть два способа, которыми IoC-контейнер разрешает зависимости: через функцию-замыкание и через автоматическое определение.

Для начала давайте посмотрим замыкания. Первым делом нечто должно быть помещёно в контейнер.

```
App::bind('foo', function($app)
{
    return new FooBar;
});
```

Извлечение из контейнера

```
$value = App::make('foo');
```

При вызове метода `App::make` вызывается соответствующее замыкание и возвращается результат её вызова.

Помещение shared ("разделяемого") типа в контейнер

Иногда вам может понадобиться поместить в контейнер тип, который должен быть создан только один раз, и чтобы все последующие вызовы возвращали бы тот же объект.

```
App::singleton('foo', function()
{
    return new FooBar;
});
```

Помещение готового экземпляра в контейнер

Вы также можете поместить уже созданный экземпляр объекта в контейнер, используя метод `instance`:

```
$foo = new Foo
App::instance('foo', $foo);
```

Где писать код ?

Вышеописанные биндинги (связывание) в IoC, как и регистраторы обработчиков событий или фильтры роутов, можно писать в старт-файле `app/start/global.php`, или, например, в файле `app/ioc.php` (имя непринципиально) и подключать его (include) в `app/start/global.php`. Если же вам ближе ООП-подход, или у вас в приложении много IoC-биндингов и вы хотите разложить их по категориям, а не держать все в одном файле, вы можете описать ваши биндинги в ваших сервис-провайдерах.

Автоопределение класса

Автоопределение класса

Контейнер IoC достаточно мощен, чтобы во многих случаях определять классы автоматически, без дополнительной настройки.

```
class FooBar {

    public function __construct(Baz $baz)
    {
        $this->baz = $baz;
    }

}

$fooBar = App::make('FooBar');
```

Обратите внимание, что даже без явного помещения класса FooBar в контейнер он всё равно был определён и зависимость Baz была автоматически внедрена в него.

Если тип не был найден в контейнере, IoC будет использовать возможности PHP для изучения класса и контроля типов в его конструкторе. С помощью этой информации контейнер сам создаёт экземпляр класса.

Связывание интерфейса и реализации

В некоторых случаях класс может принимать экземпляр интерфейса, а не сам объект. В этом случае нужно использовать метод `App::bind` для извещения контейнра о том, какая именно зависимость должна быть внедрена.

```
App::bind('UserRepositoryInterface', 'DbUserRepository');
```

Теперь посмотрим на следующий контроллер:

```
class UserController extends BaseController {

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

}
```

Благодаря тому, что мы связали `UserRepositoryInterface` с "настоящим" классом, `DbUserRepository`, он будет автоматически встроен в контроллер при его создании.

Практическое использование

Laravel предоставляет несколько возможностей для использования контейнера IoC для повышения гибкости и тестируемости вашего приложения. Основной пример - внедрение зависимости при использовании контроллеров. Все контроллеры извлекаются из IoC, что позволяет вам использовать зависимости на основе контроля типов в их конструкторах - ведь они будут определены автоматически.

Контроль типов для указания зависимостей контроллера

```
class OrderController extends BaseController {

    public function __construct(OrderRepository $orders)
    {
        $this->orders = $orders;
    }

    public function getIndex()
    {
        $all = $this->orders->all();

        return View::make('orders', compact('all'));
    }

}
```

В этом примере класс `OrderRepository` автоматически встроится в контроллер. Это значит, что при использовании юнит-тестов класс-заглушка ("mock") для `OrderRepository` может быть добавлен в контейнер, таким образом легко имитируя взаимодействие с БД.

Другие примеры использования IoC

Фильтры, вью-композеры и обработчики событий могут также извлекаться из IoC. При регистрации этих объектов

просто передайте имя класса, который должен быть использован:

```
Route::filter('foo', 'FooFilter');

View::composer('foo', 'FooComposer');

Event::listen('foo', 'FooHandler');
```

Сервис-провайдеры

Сервис-провайдеры (service providers, "поставщики услуг") - отличный способ группировки схожих регистраций в IoC в одном месте. Их можно рассматривать как начальный запуск компонентов вашего приложения. Внутри поставщика услуг вы можете зарегистрировать драйвер авторизации, классы-хранилища вашего приложения или даже собственную консольную Artisan-команду.

БОльшая часть компонентов Laravel включает в себя свои сервис-провайдеры. Все зарегистрированные сервис-провайдеры в вашем приложении указаны в массиве `providers` файла настроек `app/config/app.php`.

Создание сервис-провайдера

Для создания нового сервис-провайдера просто наследуйте класс `Illuminate\Support\ServiceProvider` и определите метод `register`.

```
use Illuminate\Support\ServiceProvider;

class FooServiceProvider extends ServiceProvider {

    public function register()
    {
        $this->app->bind('foo', function()
        {
            return new Foo;
        });
    }

}
```

Заметьте, что внутри метода `register` IoC-контейнер приложения доступен в свойстве `$this->app`. Как только вы создали провайдера и готовы зарегистрировать его в своём приложении, просто добавьте его в массиве `providers` файла настроек `app/config/app.php`.

Регистрация поставщика услуг во время выполнения

Вы можете зарегистрировать сервис-провайдер "на лету", используя метод `App::register`:

```
App::register('FooServiceProvider');
```

События контейнера

Регистрация обработчика события

IoC-контейнер запускает событие каждый раз при извлечении объекта. Вы можете отслеживать его с помощью метода `resolving`:

```
App::resolvingAny(function($object)
{
    //
});

App::resolving('foo', function($foo)
{
    //
});
```

Созданный объект передаётся в функцию-замыкание в виде параметра.

Локализация

Введение

Класс Lang даёт возможность удобного получения языковых строк, позволяя вашему приложению поддерживать несколько языков интерфейса.

Языковые файлы

Языковые строки хранятся в папке app/lang Внутри неё должны располагаться подпапки - языки, поддерживаемые приложением:

```
/app
    /lang
        /en
            messages.php
        /es
            messages.php
```

Пример языкового файла

Языковые файлы (скрипты) просто возвращают массив пар ключ/значение.

```
<?php

return array(
    'welcome' => 'Добро пожаловать на мой сайт!'
);
```

Изменение языка по умолчанию "на лету"

Язык по умолчанию указан в файле настроек app/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 app/config/app.php:

```
'fallback_locale' => 'en',
```

Основы использования

Получение строк из языкового файла

```
echo Lang::get('messages.welcome');
```

Первый компонент, передаваемый методу get - имя языкового файла, а затем 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 свы 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 `app/lang/packages/{locale}/{package}`. Например, если вам надо перекрыть файл `messages.php` пакета `skyrim/hearthfire`, путь до вашего файла локализации должен выглядеть так: `app/lang/packages/en/hearthfire/messages.php`. Нет нужды дублировать файл целиком, можно указать только те ключи, которые должны быть перекрыты.

Работа с e-mail

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`.

Основы использования

Метод `Mail::send` используется для отправки сообщения:

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Первый параметр - имя шаблона, который должен использоваться для текста сообщения. Второй - 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));
```

> **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

Помещение сообщения в очередь отправки

Из-за того, что отправка множества писем может сильно повлиять на время отклика приложения, многие разработчики помещают их в очередь на отправку. Laravel позволяет делать это, используя единое API очередей. Для помещения сообщения в очередь просто используйте метод `Mail::queue()`:

```
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 `app/config/mail.php`. Когда это сделано, сообщения будут записываться в файл журнала вашего приложения, вместо того, чтобы быть отправленными получателю.

Включение симуляции отправки

```
Mail::pretend();
```

Разработка пакетов

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. В предыдущей версии Laravel такие пакеты назывались "bundles". Они могли содержать роуты, контроллеры, шаблоны, настройки и миграции, специально рассчитанные для улучшения приложения на Laravel. 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.

Создание пакета

Простейший способ создать пакет для использования в Laravel - с помощью команды `workbench` инструмента командной строки Artisan. Сперва вам нужно установить несколько параметров в файле `app/config/workbench.php`. Там вы найдёте такие настройки `name` и `email`. Их значения будут использованы при генерации `composer.json` для вашего нового пакета. Когда вы заполнили эти значения, то всё готово для создания заготовки.

Использование команды Workbench

```
php artisan workbench vendor/package --resources
```

Имя поставщика (vendor) - способ отличить ваши собственные пакеты от пакетов других разработчиков. К примеру, если я, Taylor Otwell (автор Laravel - прим. пер.), хочу создать новый пакет под названием "Zapper", то имя поставщика может быть `Taylor`, а имя пакета - `Zapper`. По умолчанию команда `workbench` сгенерирует заготовку в общепринятом стиле пакетов, однако команда `resources` может использоваться для генерации специфичных для Laravel папок, таких как `migrations`, `views`, `config` и прочих.

Когда команда `workbench` была выполнена, ваш пакет будет доступен в папке `workbench` текущей установки Laravel. Дальше вам нужно зарегистрировать сервис-провайдер пакета, который создан автоматически предыдущей командой. Это можно сделать, добавив его к массиву `providers` файла `app/config/app.php`. Это укажет Laravel, что пакет должен быть загружен при запуске приложения. Имена классов поставщики услуг следуют схеме `[Package]ServiceProvider`. Таким образом, в примере выше мы должны были бы добавить `Taylor\Zapper\ZapperServiceProvider` к массиву `providers`.

Как только поставщик зарегистрирован вы готовы к началу разработки. Однако, перед этим, рекомендуется ознакомиться с материалом ниже, чтобы узнать о структуре пакетов и процессом их разработки.

> **Примечание:** если ваш сервис-провайдер не может быть найден, выполните команду `php artisan dump-autoload` из корня вашего приложения.

Структура пакетов

При использовании команды `workbench` ваш пакет будет настроен согласно общепринятым нормам, что позволит ему успешно интегрироваться с другими частями Laravel.

Базовая структура папок внутри пакета

```
/src
    /Vendor
        /Package
            PackageServiceProvider.php
    /config
    /lang
    /migrations
    /views
/tests
/public
```

Давайте познакомимся с этой структурой поближе. Папка `src/Vendor/Package` - хранилище всех классов вашего пакета, включая `ServiceProvider`. Папки `config`, `lang`, `migrations` и `views` содержат соответствующие ресурсы для вашего пакета (настройки, файлы локализации, миграции и шаблоны).

Сервис-провайдеры

Сервис-провайдеры - классы первичной инициализации пакета. По умолчанию они могут содержать два метода: `boot` и `register`. Внутри этих методов вы можете выполнять любой код - подключать файл с роутами, регистрировать ключи в контейнере IoC, устанавливать обработчики событий или что-то ещё.

Метод `register` вызывается сразу после регистрации сервис-провайдером, тогда как команда `boot` вызывается только перед тем, как будет обработан запрос. Таким образом, если вашему сервис-провайдеру требуется другой сервис-провайдер, который уже был зарегистрирован, или вы перекрываете другой сервис-провайдер - вам нужно использовать метод `boot`.

При создании пакета с помощью команды `workbench`, метод `boot` уже будет содержать одно действие:

```
$this->package('vendor/package');
```

Этот метод позволяет Laravel определить, как правильно загружать шаблоны, настройки и другие ресурсы вашего приложения. Обычно вам не требуется изменять эту строку, так как она настраивает пакет в соответствии с обычными нормами.

По умолчанию, после регистрации пакета `vendor/package`, его ресурсы доступны по имени `package`. Вы можете изменить это поведение:

```
// Хочу вместо `package` обращаться к ресурсам пакета по имени `custom-namespace`
$this->package('vendor/package', 'custom-namespace');

// Загрузить шаблон пакета по новому имени
$view = View::make('custom-namespace::foo');
```

Если вы хотите изменить папки, где располагаются ресурсы пакета (папки `config`, `lang`, `migrations`, `views`), вы можете указать путь до этого места третьим аргументом:

```
$this->package('vendor/package', null, '/path/to/resources');
```

Отложенные сервис-провайдеры

Если вы пишете сервис-провайдер, который не регистрирует ни один из ресурсов, то вы можете сделать его "отложенным". Такие сервис-провайдеры загружаются и регистрируются только тогда, когда сервисы, их использующие, "достаются"" из IoC-контейнера.

Для того, чтобы сделать сервис-провайдер отложенным, установите свойство `$defer` в `true`:

```
protected $defer = true;
```

Далее, вам нужно указать, каким объектам из IoC-контейнера нужен этот сервис-провайдер. Для этого создайте метод `provides`, который возвращает массив ключей, под которыми объекты регистрируются в IoC-контейнере. Например, ваш сервис-провайдер регистрирует в IoC-контейнер объекты с именами `package.service` и `package.another-service`. Соответственно, метод должен выглядеть так:

```
public function provides()
{
    return array('package.service', 'package.another-service');
}
```

Соглашения

При использовании ресурсов из пакета (конфиги или шаблоны) для отделения имени пакета обычно используют двойное двоеточие (::).

Загрузка шаблона из пакета

```
return View::make('package::view.name');
```

Чтение параметров конфига в пакете

```
return Config::get('package::group.option');
```

> **Примечание:** если ваш пакет содержит миграции, подумайте о том, чтобы сделать префикс к имени миграции - для предотвращения возможных конфликтов с именами классов в других пакетах.

Процесс разработки

При разработке пакета как правило удобно работать в контексте приложения. Для начала сделайте чистую установку Laravel, затем используйте команду `workbench` для создания структуры пакета.

После того как пакет создан, вы можете сделать `git init` изнутри папки `workbench/[vendor]/[package]`, а затем - `git push` для отправки пакета напрямую в хранилище (например, github). Это позвоит вам удобно работать в среде приложения без необходимости постоянно выполнять команду `composer update`.

Теперь, как ваши пакеты расположены в папке `workbench`, у вас может возникнуть вопрос: как Composer знает, каким образом загружать эти пакеты? Laravel автоматически сканирует папку `workbench` на наличие пакетов, загружая их файлы при запуске приложения.

Если вам нужно зарегистрировать файлы автозагрузки вашего пакета, можно использовать команду `php artisan dump-autoload`. Эта команда пересоздаст файлы автозагрузки для корневого приложения, а также всех пакетов в `workbench`, которые вы успели создать.

Выполнение команды автозагрузки

```
php artisan dump-autoload
```

Маршрутизация в пакетах

В предыдущей версии Laravel для указания URL, принадлежащих пакету, использовался параметр `handles`. Начиная с Laravel 4 пакеты могут обрабатывать любой URI. Для загрузки файлов с роутами просто подключите его через `include` из метода `boot` сервис-провайдера пакета.

Подключение файла с маршрутами из сервис-провайдера

```
public function boot()
{
    $this->package('vendor/package');

    include __DIR__.'/../../routes.php';
}
```

> **Примечание:** Если ваш сервис-провайдер использует контроллеры, вам нужно убедиться, что они верно настроены в секции автозагрузки вашего файла `composer.json`.

Конфиги в пакетах

Чтение конфига пакета

Некоторые пакеты могут требовать файлов настройки (config). Эти файлы должны быть определены аналогично файлам настроек обычного приложения. И затем, при использовании стандартной команды для регистрации ресурсов пакета `$this->package`, они будут доступны через обычный синтаксис с двойным двоеточием (::).

```
Config::get('package::file.option');
```

Однако если ваш пакет содержит всего один файл с настройками, вы можете назвать его `config.php`. Когда это сделано, то его параметры можно становятся доступными напрямую, без указания имени файла.

```
Config::get('package::option');
```

Ручная регистрация пространства имён ресурсов

Иногда вам может быть нужно зарегистрировать ресурсы пакета вне обычного вызова `$this->package`. Обычно это требуется, если ресурс расположен не в стандартном месте. Для регистрации ресурса вручную вы можете использовать методы `addNamespace` классов `View`, `Lang` и `Config`.

```
View::addNamespace('package', __DIR__.'/path/to/views');
```

Как только пространство имён было зарегистрировано, вы можете использовать его имя и двойное двоеточие для получения доступа к ресурсам:

```
return View::make('package::view.name');
```

Параметры методов `addNamespace` одинаковы для классов `View`, `Lang` и `Config`.

Редактирование конфигов пакета

Когда другие разработчики устанавливают ваш пакет им может потребоваться изменить некоторые из настроек. Однако если они сделают это напрямую в коде вашего пакета, изменения будут потеряны при следующем обновлении пакета через Composer. Вместо этого нужно использовать команд `config:publish` интерфейса Artisan.

```
php artisan config:publish vendor/package
```

Эта команда скопирует файлы настроек вашего приложения в папку `app/config/packages/vendor/package`, где разработчик может их безопасно изменять.

> **Примечание:** Разработчик также может создать настройки, специфичные для каждой среды, поместив их в `app/config/packages/vendor/package/environment`.

Миграции пакетов

Создание миграций для пакета в `workbench`

Вы можете легко создавать и выполнять миграции для любого из ваших пакетов. Для создания миграции в `workbench` используется команда `--bench` option:

```
php artisan migrate:make create_users_table --bench="vendor/package"
```

Выполнение миграций пакета в `workbench`

```
php artisan migrate --bench="vendor/package"
```

Выполнение миграций установленного пакета:

Для выполнения миграции для законченного пакета, который был установлен через Composer в папку `vendor`, вы можете использовать ключ `--package`:

```
php artisan migrate --package="vendor/package"
```

Статические файлы пакетов

Перемещение ресурсов пакета в папку `public`

Некоторые пакеты могут содержать внешние ресурсы, такие как JavaScript-код, CSS и изображения. Однако мы не можем обращаться к ним напрямую через папки `vendor` и `workbench`, так как они находятся ниже DOCROOT сервера, поэтому нам нужно перенести их в папку `public` нашего приложения. Это делает команда `asset:publish`.

```
php artisan asset:publish
```

```
php artisan asset:publish vendor/package
```

Если пакет находится в `workbench`, используйте ключ `--bench`:

```
php artisan asset:publish --bench="vendor/package"
```

Эта команда переместит ресурсы в `public/packages` в соответствии с именем поставщика и пакета. Таким образом, внешние ресурсы пакета `userscape/kudos` будут помещены в папку `public/packages/userscape/kudos`. Соблюдение этого соглашения о путях позволит вам использовать их в HTML-коде шаблонов вашего пакета.

Публикация пакетов

Когда ваш пакет готов, вы можете отправить его в Packagist. Если пакет предназначен для Laravel рекомендуется добавить тег `laravel` в файл `composer.json` вашего пакета.

Также обратите внимание, что полезно присваивать вашим выпускам номера (tags), позволяя другим разработчикам использовать стабильные версии в их файлах `composer.json`. Если стабильный выпуск ещё не готов, можно добавить директиву Composer `branch-alias`.

Когда ваш пакет опубликован, вы можете свободно продолжать его разработку в среде вашего приложения, созданной командой `workbench`. Это отличный способ, позволяющий удобно продолжать над ним работу даже после публикации .

Некоторые организации создают собственные частные хранилища пакетов для своих сотрудников. Если вам это требуется, изучите документацию проекта Satis, созданного командой разработчиков Composer.

Пагинация

Настройка

В других фрейморках пагинация (постраничный вывод данных) может быть большой проблемой. Laravel же делает этот процесс безболезненным. В файле настроек `app/config/view.php` есть единственный параметр `pagination`, который указывает, какой шаблон (views) нужно использовать при создании навигации по страницам. Изначально Laravel включает в себя два таких шаблона.

Шаблон `pagination::slider` выведет "умный" список страниц в зависимости от текущего положения, а шаблон `pagination::simple` просто создаст ссылки "Назад" и "Вперёд" для простой навигации. **Оба шаблона совместимы с Twitter Bootstrap.**

Использование

Есть несколько способов разделения данных на страницы. Самый простой - используя метод `paginate` объекта [построителя запросов](/queries) или модели Eloquent.

Постраничный вывод выборки из БД

```
$users = DB::table('users')->paginate(15);
```

Постраничный вывод запроса Eloquent

```
$allUsers = User::paginate(15);

$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 `links`:

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->links(); ?>
```

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 урлу в виде строки запроса с параметром `page`: `?page=N`.

Вы можете получить информацию о текущем положении с помощью этих методов:

- `getCurrentPage`
- `getLastPage`
- `getPerPage`
- `getTotal`
- `getFrom`
- `getTo`

Создание пагинации вручную

```
Иногда вам может потребоваться создать объект пагинации вручную. Вы можете сделать это методом `Paginator::make`:

$paginator = Paginator::make($items, $totalItems, $perPage);
```

Настройка URI для вывода ссылок

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

$users->setBaseUrl('custom/url');
```

Пример выше создаст ссылки наподобие: 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(array('sort' => 'votes'))->links(); ?>
```

Код выше создаст ссылки наподобие http://example.com/something?page=2&sort=votes

Чтобы добавить к урлу хэш-последовательность ("#xyz" в конце урла), используйте метод `fragment`:

```
<?php echo $users->fragment('foo')->links(); ?>
```

Код выше создаст ссылки типа http://example.com/something?page=2#foo

Конвертация To JSON

Класс `Paginator` реализует (implements) `Illuminate\Support\Contracts\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`, `last_page`, `from`, `to`.

Изменение отображения

По умолчанию пагинация в Laravel совместима с Twitter Bootstrap. Если вы хотите изменить html-код ссылок пагинации, вам нужно использовать свой презентер.

Расширение абстрактного презентера

Допустим, наш проект построен на css-фреймворке Zurb Foundation. Расширим (extends) `Illuminate\Pagination\Presenter` и реализуем его абстрактные методы:

```
class ZurbPresenter extends Illuminate\Pagination\Presenter {

    public function getActivePageWrapper($text)
    {
        return '<li class="current">'.$text.'</li>';
    }

    public function getDisabledTextWrapper($text)
    {
        return '<li class="unavailable">'.$text.'</li>';
    }

    public function getPageLinkWrapper($url, $page)
    {
        return '<li><a href="'.$url.'">'.$page.'</a></li>';
    }

}
```

Использование своего презентера

1. Создаем шаблон в `app/views`, который должен отображать ссылки пагинации, с таким, например, содержимым:

   render(); ?>

2. В конфиге `app/config/view.php` указываем имя этого шаблона (параметр `'pagination'`).

Очереди

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 `app/config/queue.php`. В нём вы найдёте настройки для драйверов-связей, которые поставляются вместе с фреймворком: Beanstalkd, IronMQ, Amazon SQS, а также синхронный драйвер (для локального использования).

Упомянутые выше драйвера имеют следующие зависимости:

- Beanstalkd: `pda/pheanstalk`
- Amazon SQS: `aws/aws-sdk-php`
- IronMQ: `iron-io/iron_mq`

Использование очередей

Добавление новой задачи в очередь

```
Queue::push('SendEmail', array('message' => $message));
```

Регистрация обработчика задачи

Первый аргумент метода `Queue::push` - имя класса, который должен использоваться для обработки задачи. Второй аргумент - массив параметров, которые будут переданы обработчику.

```
class SendEmail {

    public function fire($job, $data)
    {
        //
    }

}
```

Заметьте, что `fire` - единственный обязательный метод этого класса; он получает экземпляр объект `Job` и массив данных, переданных при добавлении задачи в очередь.

Задача с произвольным методом-обработчиком

Если вы хотите использовать какой-то другой метод вместо `fire` - передайте его имя при добавлении задачи.

```
Queue::push('SendEmail@send', array('message' => $message));
```

Добавление задачи в определенную очередь

У приложения может быть несколько очередей. Чтобы поместить задачу в определенную очередь, укажите её имя третьим аргументом:

```
Queue::push('SendEmail@send', array('message' => $message), 'emails');
```

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('SendEmail', 'NotifyUser'), $payload);
```

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, 'SendEmail@send', array('message' => $message));
```

Здесь для задания временного периода используется библиотека для работы с временем и датой Carbon, но `$date` может быть и просто целым числом секунд.

Удаление выполненной задачи

Как только вы закончили обработку задачи она должна быть удалена из очереди - это можно сделать методом `delete` объекта `Job`.

```
public function fire($job, $data)
{
    // Обработка задачи

    $job->delete();
}
```

Помещение задачи обратно в очередь

Если вы хотите поместить задачу обратно в очередь - используйте метод `release`:

```
public function fire($job, $data)
{
    // Обработка задачи

    $job->release();
}
```

Вы также можете указать число секунд, после которого задача будет помещена обратно:

```
$job->release(5);
```

Получение числа попыток запуска задания

Если во время обработки задания возникнет исключение (exception), задание будет помещено обратно в очередь. Вы можете получить число сделанных попыток запуска задания методом `attempts`:

```
if ($job->attempts() > 3)
{
    //
}
```

Получение идентификатора задачи

Вы также можете получить идентификатор задачи:

```
$job->getJobId();
```

Добавление функций-замыканий в очередь

Вы можете помещать в очередь и функции-замыкания. Это очень удобно для простых задач, для которых долго писать отдельный класс.

Добавить функцию-замыкание в очередь

```
Queue::push(function($job) use ($id)
{
    Account::delete($id);

    $job->delete();
});
```

> **Примечание:** Избегайте передавать при помощи `use()` в функцию-замыкание сложных объектов, например, экземпляров моделей. Передавайте вместо этого id, а экземпляры создавайте уже внутри функции-замыкания. Так вы избежите лишних ошибок при десериализации этой функции.

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 "слушает" очередь и выполняет новые задачи по мере их поступления (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 в штатном режиме, без задержек.

Обработка только первой задачи в очереди

Для обработки только одной (первой) задачи можно использовать команду `queue:work`:

```
php artisan queue:work
```

Push-очереди

Push-очереди дают вам доступ ко всем мощным возможностям, предоставляемым подсистемой очередей Laravel 4 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.

Незаконченные 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 помещается в базу данных, в таблицу `failed_jobs`. Изменить название таблицы вы можете в конфиге `app/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)
{
    //
});
```

Список всех незаконченных 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 незавершенных, используйте `queue:forget`:

```
php artisan queue:forget 5
```

Чтобы очистить весь список незавершенных задач, используйте `queue:flush`:

```
php artisan queue:flush
```

Безопасность

Конфигурация

Laravel стремится сделать реализацию авторизации максимально простой. Фактически, после установки фреймворка почти всё уже настроено. Настройки хранятся в файле `app/config/auth.php`, который содержит несколько хорошо документированных параметров для настройки поведения методов аутентификации.

"Из коробки" приложение Laravel включает в себя модель `User` в папке `app/models`, которая может использоваться вместе с дефолтным драйвером аутентификации Eloquent. При создании таблицы для данной модели убедитесь, что поле пароля принимает как минимум 60 символов.

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

> **Примечание:** Перед тем как начать, пожалуйста, убедитесь, что таблица `users` (или другая, в которой хранятся пользователи) содержит nullable (с возможностью содержать NULL) столбец `remember_token` длиною 100 символов (VARCHAR(100)). Этот столбец испольдуется для хранения токена, когда пользователь при логине ставит галку "запомнить меня".

Хранение паролей

Класс `Hash` содержит методы для безопасного хэширования с помощью Bcrypt.

Хэширование пароля по алгоритму Bcrypt:

```
$password = Hash::make('secret');
```

Проверка пароля по хэшу:

```
if (Hash::check('secret', $hashedPassword))
{
    // Пароль подходит
}
```

Проверка на необходимость перехэширования пароля:

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

Аутентификация пользователей

Для аутентификации пользователя в вашем приложении вы можете использовать метод `Auth::attempt`.

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{
    return Redirect::intended('dashboard');
}
```

Заметьте, что поле `email` не обязательно и оно используется только для примера. Вы должны использовать любое поле, которое соответствует имени пользователя в вашей БД. Метод `Redirect::intended` отправит пользователя на URL, который он пытался просмотреть до того, как запрос был перехвачен фильтром аутентификации. Также в этом методе можно задать дополнительный URL, куда будет осуществлен переход, если первый URL не доступен.

Когда вызывается метод `attempt`, запускается событие `auth.attempt`. При успешной авторизации также запускается событие `auth.login`.

Проверка авторизации пользователя

Для определения того, авторизован ли пользователь или нет, можно использовать метод check.

```
if (Auth::check())
{
    // Пользователь уже вошёл в систему
}
```

Если вы хотите предоставить функциональность типа "запомнить меня", то вы можете передать `true` вторым параметром к методу `attempt`, который будет поддерживать авторизацию пользователя без ограничения по времени (пока он вручную не выйдет из системы). В такоем случае у вашей таблицы `users` должен быть строковой столбец `remember_token` для хранения токена пользователя.

```
if (Auth::attempt(array('email' => $email, 'password' => $password), true))
{
    // Пользователь был запомнен
}
```

Примечание: если метод `attempt` вернул `true`, то пользователь успешно вошёл в систему.

Имеет ли пользователь токен "запомнить меня"

Метод `viaRemember` позволяет узнать, вошел ли пользователь при помощи фичи "запомнить меня".

```
if (Auth::viaRemember())
{
    // Пользователь вошел, так как ранее ставил галку "запомнить меня"
}
```

Авторизация пользователя с использованием условий

Вы также можете передать дополнительные условия для запроса к таблице:

```
if (Auth::attempt(array('email' => $email, 'password' => $password, 'active' => 1)))
{
    // Вход, если пользователь активен, не отключен и существует.
}
```

> **Примечание** Для повышения безопасности после аутентификации фреймворк регенерирует ID сессии пользователя.

Доступ к залогиненному пользователю

Как только пользователь авторизован вы можете обращаться к модели `User` и её свойствам.

```
$email = Auth::user()->email;
```

Для простой аутентификации пользователя по ID используется метод `loginUsingId`:

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

Проверка данных для входа без авторизации

Метод `validate` позволяет вам проверить данные для входа без осуществления самого входа.

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

Аутентификация пользователя на один запрос

Вы также можете использовать метод once для авторизации пользователя в системе только для одного запроса. Сессии и cookies не будут использованы.

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

Выход пользователя из приложения

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

Ручная авторизация

Если вам нужно войти существующим пользователем, просто передайте его модель в метод `login`:

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

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

Это эквивалентно аутентификации пользователя через его данные методом `attempt`.

Аутентификация и роутинг

Вы можете использовать Фильтры роутов, чтобы позволишь только залогиненным пользователям обращаться к заданному роуту или группе роутов. Изначально Laravel содержит фильтр `auth`, который содержится в файле `app/filters.php`.

Защита роута аутентификацией

```
Route::get('profile', array('before' => 'auth', function()
{
    // Доступно только залогиненным пользователям.
}));
```

Защита от подделки запросов (CSRF)

Laravel предоставляет простой способ защиты вашего приложения от подделки межсайтовых запросов (cross-site request forgeries, CSRF).

Вставка CSRF-ключа в форму

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

Проверка переданного CSRF-ключа

```
Route::post('register', array('before' => 'csrf', function()
{
    return 'Вы передали верный ключ!';
}));
```

HTTP-аутентификация

HTTP Basic Authentication - простой и быстрый способ аутентификации пользователей вашего приложения без создания дополнительной страницы входа. При HTTP-аутентификации форму для ввода логина и пароля показывает браузер, в виде всплывающего окна. Для HTTP-аутентификации используйте фильтр `auth.basic`:

Защита роута фильтром HTTP-аутентификации

```
Route::get('profile', array('before' => 'auth.basic', function()
{
    // Доступно только залогиненным пользователям.
}));
```

По умолчанию, фильтр `basic` будет использовать поле `email` модели объекта при аутентификации. Если вы хотите использовать иное поле, можно передать его имя первым параметром методу `basic` в файле `app/filters.php`:

```
Route::filter('auth.basic', function()
{
    return Auth::basic('username');
});
```

Авторизация без запоминания состояния

Вы можете использовать HTTP-авторизацию без установки cookies в сессии, что особенно удобно для аутентификации в API. Для этого зарегистрируйте фильтр, вовращающий результат вызова `onceBasic`.

```
Route::filter('basic.once', function()
{
    return Auth::onceBasic();
```

```
});
```

Если вы используете Apache + PHP FastCGI, HTTP-аутентификация не будет работать "из коробки". Вам нужно добавить следующие строки в свой `.htaccess`:

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

Сброс забытого пароля

Модель и таблица

Восстановление забытого пароля - очень распространенная вещь в веб-приложениях. Чтобы не заставлять вас писать его вновь и вновь, Laravel предоставляет встроенные удобные методы для совершения подобных операций. Для начала убедитесь, что ваша модель `User` реализует (implements) интерфейс `Illuminate\Auth\Reminders\RemindableInterface`. Модель `User`, которая идет с фреймворком, уже реализует его.

Реализация RemindableInterface

```
class User extends Eloquent implements RemindableInterface {

    public function getReminderEmail()
    {
        return $this->email;
    }
}
```

Создание таблицы токенов сброса пароля

Далее, должна быть создана таблица для хранения токенов запросов сброса пароля. Для создания такой таблицы существует artisan-команда `auth:reminders-table`.

```
php artisan auth:reminders-table

php artisan migrate
```

Контроллер восстановления пароля

Чтобы автоматически создать контроллер восстановления пароля, воспользуйтесь командой `auth:reminders-controller`. В папке `controllers` будет создан `RemindersController.php`:

```
php artisan auth:reminders-controller
```

Созданный контроллер содержит метод `getRemind`, который показывает форму для напоминания пароля. Вам надо создать эту форму во вьюхе `password.remind` (файл remind.blade.php в папке views/password - см. view). Форма должна отправлять POST с email на метод `postRemind`

Простейший пример `password.remind`:

```
<form action="{{ action('RemindersController@postRemind') }}" method="POST">
    <input type="email" name="email">
    <input type="submit" value="Send Reminder">
</form>
```

Метод `postRemind` уже есть в сгенеренном `RemindersController.php`. Приняв `email` POST-запросом, контроллер отсылает на этот адрес письмо с подтверждением. Если оно отсылается нормально, в сессию во flash-переменную `status` заносится сообщение об успешной отправке. Если нет - во flash-переменную `error` заносится текст ошибки.

Для модификации собщения, которое уйдет пользователю на почту, вы можете изменить в контроллере вызов `Password::remind` на, например, такое:

```
Password::remind(Input::only('email'), function($message)
{
    $message->subject('Password Reminder');
});
```

Пользователь получит письмо со ссылкой на метод `getReset`, с токеном для индентификации пользователя. Этот метод вызывает вьюху `password.reset` (файл `reset.blade.php` в папке `views/password`), в которой должна быть форма для смены пароля, со скрытым полем `token` и полями `email`, `password`, and `password_confirmation`, например такая:

```
<form action="{{ action('RemindersController@postReset') }}" method="POST">
```

```
    <input type="hidden" name="token" value="{{ $token }}">
    <input type="email" name="email">
    <input type="password" name="password">
    <input type="password" name="password_confirmation">
    <input type="submit" value="Reset Password">
</form>
```

Метод `postReset` производит замену паролей в вашем сторадже. По умолчанию считается, что пользователи хранятся в БД, с которой умеет работать Eloquent - ожидается $user, который передается в функцию-замыкание имеет метод `save`. Если это не так, измените функцию-замыкание в аргументе `Password::reset` в контроллере `RemindersController` исходя из вашей архитектуры приложения.

Если смена пароля прошла удачно, пользователь редиректится на главную страницу вашего приложения (вы можете изменить URL редиректа, если хотите). Если нет - пользователь редиректится обратно с установкой flash-переменной `error`.

Валидация паролей

По умолчанию `Password::reset` валидирует пароль пользователя исходя из двух правил - введенные пароли должны совпадать и пароль должен быть больше или равен 6 символам. Если вы хотите расширить валидацию паролей, вы можете определить свой `Password::validator`:

```
Password::validator(function($credentials)
{
    return strlen($credentials['password']) >= 6;
});
```

> **Примечание** Токены сброса пароля валидны в течении одного часа. Вы можете изменить это в `app/config/auth.php`, параметр `reminder.expire`.

Шифрование

Laravel предоставляет функции устойчивого шифрования по алгоритму AES с помощью расширения `mcrypt` для PHP.

Шифрование строки

```
$encrypted = Crypt::encrypt('секрет');
```

> **Примечание:** обязательно установите 16, 24 или 32-значный ключ `key` в файле `app/config/app.php`. Если этого не сделать, зашифрованные строки не будут достаточно надежными.

Расшифровка строки

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Изменение режима и алгоритма шифрования

Вы также можете установить свой алгоритм и режим шифрования:

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

Драйвера аутентификации

Laravel из коробки предлагает для аутентификации драйвера `database` и `eloquent`. Чтобы узнать больше о том, как добавлять свои драйвера, изучите документацию по расширению системы аутентификации.

Сессии

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 `app/config/session.php`. Обязательно просмотрите параметры, доступные вам - они хорошо документированы. По умолчанию Laravel использует драйвер `native`, который подходит для большинства приложений.

Зарезервированные ключи

Не используйте в качестве ключа переменной `flash` - этот клюс зарезервирован Laravel для внутренних нужд.

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'; });
```

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 - а дальше 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 команду `session:table` Artisan 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.

SSH

Настройка

В состав Laravel входит библиотека для коннекта к серверам по ssh и исполнения там команд, что позволяет писать artisan-команды для работы на удаленных серверах. Для работы с этой библиотекой Laravel предоставляет фасад SSH.

Файл настроек этой библиотеки - `app/config/remote.php`. Массив `connections` содержит список доступных серверов. Доступ к серверу может осуществляться по паролю или ключу.

> **Примечание:** Если вам надо запускать разнообразные задачи на своем сервере, попробуйте Envoy.

Использование ssh

Запуск команды на дефолтном сервере

```
SSH::run(array(
    'cd /var/www',
    'git pull origin master',
));
```

Запуск команды на указанном сервере

```
SSH::into('staging')->run(array(
    'cd /var/www',
    'git pull origin master',
));
```

Получение ответа

Вы можете ловить текстовый вывод исполненных команд функцией-замыканием, переданной вторым аргументом:

```
SSH::run($commands, function($line)
{
    echo $line.PHP_EOL;
});
```

Задачи

Вы можете объединять несколько команд в т.н. задачу:

```
SSH::into('staging')->define('deploy', array(
    'cd /var/www',
    'git pull origin master',
    'php artisan migrate',
));
```

Когда задача задана, вы можете исполнить её:

```
SSH::into('staging')->task('deploy', function($line)
{
    echo $line.PHP_EOL;
});
```

SFTP загрузка

Класс SSH позволяет загрузить файл по SFTP, при помощи методов `get` и `getString`.

```
SSH::into('staging')->get($remotePath, $localPath);

$contents = SSH::into('staging')->getString($remotePath);
```

SFTP аплоад

Таже вы можете закачивать файлы на удаленный сервер по SFTP:

```
SSH::into('staging')->put($localFile, $remotePath);

SSH::into('staging')->putString($remotePath, 'Foo');
```

Показ логов

Laravel имеет удобное средство для просмотра последних изменений лог-файлов на удаленных серверах - т.н. tailing, когда на экран выводится только то, что добавилось в конец файла. Просто укажите после artisan-команды tail имя удаленного сервера.

```
php artisan tail staging

php artisan tail staging --path=/path/to/log.file
```

Envoy

Envoy - это инструмент для запуска задач на удаленных серверах. Он предоставляет простой синтаксис для записи операций деплоя, запуска artisan-команд и т.п., который базируется на синтаксисе Blade.

> **Примечание:** Envoy требует PHP 5.4 и выше, запускается на Mac или Linux.

Установка

1. Установите Envoy глобально в системе:

   composer global require "laravel/envoy=~1.0"

Проверьте, чтобы `~/.composer/vendor/bin` был у вас в PATH. Для проверки наберите `envoy` в терминале - должен вывестись краткий хелп.

1. Создайте `Envoy.blade.php` в корне вашего проекта. Например, задача ожет быть такой:

   @servers(['web' => '192.168.1.1'])

   @task('foo', ['on' => 'web']) ls -la @endtask

Директива `@servers` задает список доступных серверов. Внутри `@task` располагаются bash-команды. На каком сервере запускать задачу говорит параметр 'on'.

Команда `init` создает заготовку envoy-файла:

```
envoy init user@192.168.1.1
```

Запуск задач

Для запуска задач служит команда `run`

```
envoy run foo
```

Вы можете передать переменную в вашу задачу:

```
envoy run deploy --branch=master
```

В самой задаче она доступна как переменная в blade-шаблоне, в фигурных скобках:

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
```

```
@endtask
```

Инициализация

Вы можете использовать директиву `@setup` для объявления переменных. Поддерживается синтаксис PHP.

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@endsetup
```

Вы также можете использовать директиву `@include` чтобы подключить любой php-файл.

```
@include('vendor/autoload.php');
```

Запуск на нескольких серверах

Вы можете запустить выполнение задачи на нескольких серверах, задав их имена в массиве 'on':

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

По умолчанию задачи выполняются последовательно. Пока не закончится выполнение задачи на одном сервере, выполнение на втором не начнется.

Параллельное выполнение

Для одновременного запуска задач, установите параметр `'parallel'` в `true`.

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Макросы

Макросы позволяют объединять несколько `@task`:

```
@servers(['web' => '192.168.1.1'])

@macro('deploy')
    foo
    bar
@endmacro

@task('foo')
    echo "HELLO"
@endtask

@task('bar')
    echo "WORLD"
@endtask
```

Полученный макрос `deploy` может быть вызван как обычная команда:

```
envoy run deploy
```

Уведомления

После выполнения задачи envoy может послать уведомление об этом.

HipChat

Для отсылки нотификации в чат групповой разработки HipChat служит директива `@hipchat`.

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask

@after
    @hipchat('token', 'room', 'Envoy')
@endafter
```

Чтобы изменить сообщение, вы можете задействовать переменные, объявленные в директиве `@setup`, или взятые из подключенного `@include` php-файла:

```
@after
    @hipchat('token', 'room', 'Envoy', "$task ran on [$environment]")
@endafter
```

Slack

Для отсылки уведомления в Slack служит директива `@slack`:

```
@after
    @slack('team', 'token', 'channel')
@endafter
```

Обновление Envoy

Для обновления Envoy используйте команду `self-update`

```
envoy self-update
```

Если вы установили envoy в `/usr/local/bin`, используйте sudo

```
sudo envoy self-update
```

Шаблоны

Сборка шаблонов в контроллере

Один из способов компоновки шаблонов в Laravel - использовать сборщик шаблонов контроллера. Если в классе контроллера определить свойство `layout`, то указанный шаблон будет создан автоматически и будет использоваться при генерации ответа клиенту.

Использование сборки шаблонов в контроллере

```
class UserController extends BaseController {

    /**
     * Шаблон, который должен использоваться при ответе.
     */
    protected $layout = 'layouts.master';

    /**
     * Отображает профиль пользователя.
     */
    public function showProfile()
    {
        $this->layout->content = View::make('user.profile');
    }

}
```

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

```
<!-- Расположен в app/views/layouts/master.blade.php -->

<html>
    <body>
        @section('sidebar')
            Это - главная боковая панель.
        @show

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

Использование шаблона Blade

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

@section('sidebar')
    @parent

    <p>Этот элемент будет добавлен к главной боковой панели.</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 }}
```

Конечно, любой вывод переменных должен быть должен быть безопасным - все управляющие символы (javascript-код) должны быть преобразованы в обычные видимые символы, во избежание разнообразных атак. Blade это делает автоматически, если переменная выводится в тройных скобках:

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

Если вы хотите отменить это, выводите переменные в двойных скобках, но будьте осторожны и не выводите в них контент, который мог изменить пользователь:

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

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

@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', array('some'=>'data'))
```

Перезапись секций

По умолчанию, содержимое новой секции добавляется в конец содержимого старой (перекрытой) секции. Для полной перезаписи 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);
```

Комментарии

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

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);
});
```

Юнит-тесты

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. В дополнение к PHPUnit Laravel также использует компоненты Symfony HttpKernel, DomCrawler и BrowserKit для тестирования ваших шаблонов при помощи эмуляции браузера.

Папка `app/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 `app/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.

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));
```

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');
```

> **Примечание:** фильтры роутов отключены в тестовой среде. Для их включения добавьте в тест вызов `Route::enableFilters()`.

Работа с DOM

Вы можете обратиться к URL и получить объект `DomCrawler`, который может использоваться для проверки содержимого ответа:

```
$crawler = $this->client->request('GET', '/');

$this->assertTrue($this->client->getResponse()->isOk());

$this->assertCount(1, $crawler->filter('h1:contains("Hello World!")'));
```

Для более подробной информации о его использовании обратитесь к официальной документации.

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', array('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(array('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.

Валидация

Использование валидации

Laravel поставляется с простой, удобной системой валидации (проверки входных данных на соответствие павилам) 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 символом вертикальной черты (|), 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.

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 ($validator->fails())
    {
        return Redirect::to('register')->withErrors($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 в 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'); ?>
```

Доступные правила проверки

Ниже список всех доступных правил и их функции:

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 только латинские символы.

alpha_dash

Поле можно содержать только латинские символы, цифры, знаки подчёркивания (_) и дефисы (-).

alpha_num

Поле можно содержать только латинские 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`.

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 или gif.

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 значение *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.

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 понадобиться 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
<?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)
{
    //
});
```

Основы работы с базой данных

Настройка

Laravel делает процесс соединенеия с БД и выполнение запросов очень простым. Настройки работы с БД хранятся в файле `app/config/database.php`. Здесь вы можете указать все используемые вами соединения к БД, а также задать то, какое из них будет использоваться по умолчанию. Примеры настройки всех возможных видов подключений находятся в этом же файле.

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

Раздельное чтение и запись

Если вы используете репликацию БД, для улучшения производительности вы можете использовать не одно, а два соединения с БД на разных серверах. С одного сервера (slave) вы можете только делать SELECT, а на другом (master) - INSERT, UPDATE и DELETE. Laravel прозрачно позволяет задействовать такой вариант работы, причем не важно, что вы используете - сырые запросы `DB::select()`, Query Builder или Eloquent ORM.

Вот пример конфигурации:

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

Вы можете располагать в `'read'` и `'write'` не только `'host'`, но и любые mysql-настройки, которые отличаются от общих настроек БД и уникальны для данного сервера - например, `'database'`, `'username'` и `'password'`. Если их не указывать, будут взяты общие настройки.

Вы также можете задать несколько серверов для операции чтения или записи. В таком случае конфиг будет выглядеть так:

```
'read' => array(
        array('host' => '192.168.1.10'),
        array('host' => '192.168.1.11'),
    ),
```

В таком случае сервер для чтения будет выбран случайным образом.

Выполнение запросов

Как только вы настроили соединение с базой данных вы можете выполнять запросы, используя класс `DB`.

Выполнение запроса SELECT

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

Метод `select` всегда возвращает массив.

Выполнение запроса INSERT

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

Выполнение запроса UPDATE

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

Выполнение запроса DELETE

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

> **Примечание:** методы `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

Вы можете использовать метод `transaction` для выполнения запросов внутри одной транзации:

```
DB::transaction(function()
{
    DB::table('users')->update(array('votes' => 1));

    DB::table('posts')->delete();
});
```

> **Транзакция** - особое состояние БД, в котором выполняемые запросы либо все вместе успешно завершаются, либо (в случае ошибки, а конкретно - возбуждении исключения (exception)) все их изменения откатываются. Это позволяет поддерживать целостность внутренней структуры данных. К примеру, если вы вставляете запись о заказе, а затем в отдельную таблицу добавляете товары, то при неуспешном выполнении скрипта (в том числе падения вёб-сервера, ошибки в запросе и пр.) СУБД автоматически удалит запись о заказе и все товары, которые вы успели добавить - прим. пер.

Также доступны ручные операции с транзакциями:

Начать транзакцию:

```
DB::beginTransaction();
```

Откатить транзакцию:

```
DB::rollback();
```

Завершить транзакцию:

```
DB::commit();
```

Другие соединения

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 и вы можете сделать это так:

```
DB::reconnect('foo');
```

Если вам нужно отключиться от БД - например, чтобы не превышать `max_connections` БД, вы можете воспользоваться методом `disconnect`:

```
DB::disconnect('foo');
```

Журнал запросов

По умолчанию Laravel записывает все SQL-запросы, выполненные в рамках текущего HTTP-запроса. Однако в некоторых случаях - например, при вставке большого набора записей - это может быть слишком ресурсозатратно. Для отключения журнала вы можете использовать метод `disableQueryLog`:

```
DB::connection()->disableQueryLog();
```

Чтобы получить массив исполненых запросов используйте метод `getQueryLog`:

```
$queries = DB::getQueryLog();
```

Конструктор запросов

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);
}
```

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

Группировка условий

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.

```
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 сырого выражения

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

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'));
```

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));
```

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 закэшировать запрос методом `remember`:

```
$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();
```

Eloquent ORM

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 `app/config/database.php`.

Использование ORM

Для начала создадим модель Eloquent. Модели обычно располагаются в папке `app/models`, но вы можете поместить в любое место, в котором работает автозагрузчик в соответствии с вашим файлом `composer.json`.

Создание модели Eloquent

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

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 Eloquent {

    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 конструктора, то можно использовать метод `whereRaw`:

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

Обработка результата по частям

Если в вашей eloquent-выборке получилось очень много элементов, то обрабатывая их в цикле foreach вы можете выйти за пределы оперативной памяти. Чтобы этого не произошло, используйте следующий трюк:

```
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);
```

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 Eloquent {

    protected $fillable = array('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 Eloquent {

    protected $guarded = array('id', 'password');

}
```

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.

> **Примечание:** Даже если вы используете `guarded`, по возможности избегайте массового присваивания `Input::get()` модели.

Защита всех атрибутов от массового заполнения

```
protected $guarded = array('*');
```

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();
```

> **Внимание:** обычно ваши модели Eloquent содержат автоматические числовые ключи (autoincrementing). 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 Eloquent {

    protected $guarded = array('id', 'account_id');

}
```

Создание модели

```
$user = User::create(array('name' => 'Джон'));
```

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

Обновление полученной модели

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

$user->email = 'john@foo.com';
```

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

Иногда вам может быть нужно сохранить не только модель, но и все её отношения. Для этого просто используйте метод push.

Сохранение модели и её отношений

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

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

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

Для удаления модели вызовите метод delete на её объекте.

Удаление существующей модели

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

$user->delete();
```

Удаление модели по ключу

```
User::destroy(1);

User::destroy(array(1, 2, 3));

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

Конечно, вы также можете выполнять удаление на наборе моделей:

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

Если вам нужно просто обновить время изменения записи - используйте метод touch:

Обновление времени изменения модели

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

Мягкое удаление

Когда вы "мягко" удаляете модель, она на самом деле остаётся в базе данных, однако устанавливается её поле deleted_at. Для включения мягких удалений на модели определите свойство её softDelete:

```
class User extends Eloquent {

    protected $softDelete = true;

}
```

Для добавления поля deleted_at к таблице можно использовать метод softDeletes из миграции:

```
$table->softDeletes();
```

Теперь когда вы вызовите метод delete, поле deleted_at будет установлено в значение текущего времени. При запросе моделей, использующих мягкое удаление, "удалённые" модели не будут включены в результат запроса. Для отображения всех моделей, в том числе удалённых, используйте метод withTrashed.

Включение удалённых моделей в результат выборки

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

Если вы хотите получить **только** удалённые модели, вызовите метод onlyTrashed:

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

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

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

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())
{
    //
}
```

Поля времени

По умолчанию Eloquent автоматически поддерживает поля `created_at` и `updated_at` . Просто добавьте эти timestamp-поля к таблице и Eloquent позаботится об остальном. Если вы не хотите, чтобы он поддерживал их, добавьте свойство `timestamps` к классу модели.

Отключение автоматических полей времени

```
class User extends Eloquent {

    protected $table = 'users';

    public $timestamps = false;

}
```

Для настройки форматов времени перекройте метод `getDateFormat`:

Использование собственного формата времени

```
class User extends Eloquent {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

Заготовки запросов

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

Создание заготовки запроса

```
class User extends Eloquent {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Использование заготовки

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

Динамические заготовки

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

```
class User extends Eloquent {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

А затем передайте их при вызове метода заготовки:

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

Отношения

Конечно, ваши таблицы скорее всего как-то связаны с другими таблицами БД. Например, статья в блоге может иметь много комментариев, а заказ может быть с связан с оставившим его пользователем. Eloquent упрощает работу и управление такими отношениями. Laravel поддерживает 4 типа связей:

- Один к одному
- Один ко многим
- Многие ко многим
- Полиморфические 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.

Создание связи "один к одному"

```
class User extends Eloquent {

    public function phone()
    {
        return $this->hasOne('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 поле в таблице называется по имени модели плюс `_id`. В данном случае предполагается, что это `user_id`.Если вы хотите перекрыть стандартное имя передайте второй параметр методу `hasOne`:

```
return $this->hasOne('Phone', 'custom_key');
```

Для создания обратного отношения в модели Phone используйте метод `belongsTo` ("принадлежит к"):

Создание обратного отношения

```
class Phone extends Eloquent {

    public function user()
    {
```

```
        return $this->belongsTo('User');
    }

}
```

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

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User', 'custom_key');
    }

}
```

Один ко многиму

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

```
class Post extends Eloquent {

    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 ключа:

```
return $this->hasMany('Comment', 'custom_key');
```

Для определения обратного отношения используйте метод `belongsTo`:

Определение обратного отношения

```
class Comment extends Eloquent {

    public function post()
    {
        return $this->belongsTo('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 Eloquent {

    public function roles()
    {
        return $this->belongsToMany('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('Role', 'user_roles');
```

Вы также можете перекрыть имена ключей по умолчанию:

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

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

```
class Role extends Eloquent {

    public function users()
    {
        return $this->belongsToMany('User');
    }

}
```

Полиморфические отношения

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

```
class Photo extends Eloquent {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}

class Order extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

Теперь мы можем получить фотографии и для персонала, 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 персонала, а второе - имя класса-модели владельца. Это позволяет ORM определить, какой класс модели должен быть возвращёт при использовании отношения `imageable`.

Запросы к отношениям

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:

Проверка связей при выборке

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

Вы также можете указать оператор и число:

```
$posts = Post::has('comments', '>=', 3)->get();
```

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". К примеру, для следующей модели $phone:

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('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;
```

Активная загрузка

Активная загрузка (eager loading) призвана устранить проблему запросов N + 1. Например, представьте, что у нас есть

модель Book со связью к модели Author. Отношение определено как:

```
class Book extends Eloquent {

    public function author()
    {
        return $this->belongsTo('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 активную загрузку для кардинального уменьшения числа запросов. Отношение будет активно загружено, если оно было указано при вызове метода with:

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

В цикле выше будут выполнены всего два запроса:

```
select * from books

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

Разумное использование активной загрузки поможет сильно повысить производительность вашего приложения.

Конечно, 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 активно загружена вместе со связью contacts модели автора.

Ограничения активной загрузки

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

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

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

Ленивая активная загрузка

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

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

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

Вставка связанных моделей

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

Создание связанной модели

```
$comment = new Comment(array('message' => 'Новый комментарий.'));

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

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

Связывание моделей (Belongs To)

При обновлении связей belongsTo ("принадлежит к") вы можете использовать метод associate. Он установит внешний ключ на связанной модели:

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

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

$user->save();
```

Связывание моделей (многие ко многим)

Вы также можете вставлять связанные модели при работе с отношениями многие ко многим. Продолжим использовать наши модели User и Role в качестве примеров. Вы можем легко привязать новые роли к пользователю методом attach.

Связывание моделей "многие ко мнгим"

```
$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, array('expires' => $expires));
```

Конечно, существует противоположность attach - detach:

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

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.

Использование Sync для привязки моделей "многие ко многим"

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

Вы также можете связать другие связующие таблицы с нужными ID.

Добавление данных для связующий таблицы при синхронизации

```
$user->roles()->sync(array(1 => array('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(array('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, array('expires' => $expires));
```

Обновление 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 Eloquent {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('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 таблицы. Например, предположим, что наш объект User имеет множество связанных объектов Role. После чтения отношения мы можем прочитать таблицу 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('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('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.

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 Eloquent {

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

}
```

Читатели и преобразователи

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 (он же - "стиль Си", с подчёркиваниями -прим. пер.).

Объявление читателя

```
class User extends Eloquent {

    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 Eloquent {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

Преобразователи дат

По умолчанию Eloquent преобразует поля `created_at`, `updated_at` и `deleted_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 array('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 врените пустой массив из метода `getDates`.

```
public function getDates()
{
    return array();
}
```

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`.

Если обработчики `creating`, `updating`, `saving` или `deleting`вернут значение `false`, то действие будет отменено.

Отмена сохранения модели через события

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

Модели Eloquent также содержат статический метод `boot`, который может быть хорошим местом для регистрации ваших обработчиков событий.

Определение метода boot

```
class User extends Eloquent {

    public static function boot()
    {
        parent::boot();
```

```
        // Регистрация ваших обработчиков...
    }

}
```

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);
```

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 Eloquent {

    protected $hidden = array('password');
```

```
}
```

Вы также можете использовать атрибут `visible` для указания разрешённых полей:

```
protected $visible = array('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'] == 'да';
}
```

Как только вы создали читателя добавьте его имя к свойству-массиву `appends` класса модели:

```
protected $appends = array('is_admin');
```

Как только атрибут был добавлен к списку `appends`, он будет включён в массивы и выражения JSON, образованные от этой модели.

Конструктор таблиц

Введение

Класс Schema представляет собой независимый от БД интерфейс манипулирования таблицами. Он хорошо работает со всеми СУБД, 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 мы будем использовать метод `Schema::table`:

```
Schema::table('users', function(Blueprint $table)
{
    $table->string('email');
});
```

Конструктор таблиц поддерживает различные типы полей:

Команда	Описание
`$table->increments('id');`	Первичный последовательный ключ (autoincrement).
`$table->bigIncrements('id');`	Первичный последовательный ключ типа BIGINT.
`$table->string('email');`	Поле VARCHAR

`$table->string('name', 100);`	Поле VARCHAR с указанной длиной
`$table->integer('votes');`	Поле INTEGER
`$table->bigInteger('votes');`	Поле BIGINT
`$table->smallInteger('votes');`	Поле SMALLINT
`$table->float('amount');`	Поле FLOAT
`$table->decimal('amount', 5, 2);`	Поле DECIMAL с указанной размерностью и точностью
`$table->boolean('confirmed');`	Поле BOOLEAN
`$table->date('created_at');`	Поле DATE
`$table->dateTime('created_at');`	Поле DATETIME
`$table->time('sunrise');`	Поле TIME
`$table->timestamp('added_on');`	Поле TIMESTAMP
`$table->timestamps();`	Добавляет поля **created_at** и **updated_at**
`$table->softDeletes();`	Добавляет поле **deleted_at** для мягкого удаления
`$table->text('description');`	Поле TEXT
`$table->binary('data');`	Поле BLOB
`$table->enum('choices', array('foo', 'bar'));`	Поле ENUM
`->nullable()`	Указывает, что поле может быть NULL
`->default($value)`	Указывает значение по умолчанию для поля
`->unsigned()`	Обозначает беззнаковое число UNSIGNED

Вставка поля после существующего (в MySQL)

$table->string('name')->after('email');

```
$table->string('name')->after('email');
```

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');
});
```

> **Внимание:** переименование полей типа `enum` не поддерживается.

Удаление полей

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('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: можно определять их во время определения полей, либо добавлять 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 ваших таблиц:

```
$table->foreign('user_id')->references('id')->on('users');
```

В этом примере мы указываем, что поле `user_id` связано с полем `id` таблицы `users`.

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"

Storage Engines

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`). Правда, использование таблиц различных архитектур в одном запросе заметно снижает его производительность - прим. пер.

Миграции и начальные данные

Введение

Миграции - это что-то вроде системы контроля версий для вашей базы данных. Они позволяют команде программистов изменять её структуру, 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 `migrate:make` командного интерфейса Artisan ("артизан-команда").

```
php artisan migrate:make create_users_table
```

Миграция будет помещена в папку `app/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.

При создании миграции вы можете также передать параметр `--path`. Путь должен быть относительным к папке вашей установки Laravel.

```
php artisan migrate:make foo --path=app/migrations
```

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 migrate:make create_users_table --table=users --create
```

Применение миграций

Накатывание всех новых неприменённых миграций

```
php artisan migrate
```

Накатывание новых миграций, расположенных в указанной папке

```
php artisan migrate --path=app/foo/migrations
```

Накатывание новых миграций для пакета

```
php artisan migrate --package=vendor/package
```

> **Внимание:** если при применении миграций вы сталкиваетесь с ошибкой "class not found" ("Класс не найден") - попробуйте выполнить команду `composer dump-autoload`.

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 app/database/seeds. Они могут иметь любое имя, но вам, вероятно, следует придерживаться какой-то логики в их именовании - например, UserTableSeederи 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
```

Redis

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.

> **Внимание:** Если у вас установлено расширение Redis через PECL, вам нужно переименовать псевдоним в файле `app/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' => array(

    'cluster' => true,

    'default' => array('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);
    }
});
```

Интерфейс Artisan

Введение

Artisan - название инструмента командной строки, с которым поставляется Laravel. Он содержит набор полезных команд, помогающие вам при разработке приложения. Он основан на мощном компоненте Symfony Console.

Использование

Список всех доступных команд

Для просмотра списка доступных команд используйте команду `list`:

```
php artisan list
```

Помощь по выбранной команде

Каждая команда также включает экран помощи, который отображает и описывает доступные параметры и ключи данной команды. Для просмотра помощи просто добавьте имя команды после слова `help`.

```
php artisan help migrate
```

Указание имени среды для выполнения команды

Вы можете указать среду, которая будет использоваться при выполнении команды, с помощью ключа `--env`:

```
php artisan migrate --env=local
```

Версия текущей установки Laravel

Вы также можете узнать версию текущей установки Laravel с помощью ключа `--version`:

```
php artisan --version
```

Разработка Artisan-команд

Введение

В дополнение к командам, предоставляемых Artisan'ом, Вы также можете создавать свои собственные команды для работы с Вашим приложением. Вы можете хранить свои команды в директории `app/commands`; однако, вы вправе сами выбирать место для хранения, убедившись, что команды могут быть автоматически загружены, основываясь на настройках в Вашем `composer.json`.

Создание Команды

Генерация Класса

Для создания новой команды, вы можете воспользоваться командой Artisana'a `command:make`, которая сгенерирует макет класса:

Сгенерируйте новый класс команды

```
php artisan command:make FooCommand
```

По умолчанию Ваши команды будут помещены в директорию `app/commands`; однако, Вы можете указать произвольный путь или пространство имен:

```
php artisan command:make FooCommand --path=app/classes --namespace=Classes
```

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

```
php artisan command:make AssignUsers --command=users:assign
```

Написание Команды

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

Метож `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 произвольные аргументы или опции, которые будет принимать Ваша команда. Оба этих метода возвращают массив команд, описываемых списокм полей массива.

Массив, определяющий аргумент, выглядит так:

```
array($name, $mode, $description, $defaultValue)
```

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

Массив, определяющий опцию, выглядит следующим образом:

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

Для опций, аргумент mode может быть: `InputOption::VALUE_REQUIRED` (значение обязательно), `InputOption::VALUE_OPTIONAL` (значение необязательно), `InputOption::VALUE_IS_ARRAY` (значение - это массив), `InputOption::VALUE_NONE` (нет значения).

Режим `VALUE_IS_ARRAY` обозначает, что этот переключатель может быть использован несколько раз при вызове команды:

```
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 исполнения 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('Отобразить это на экране');
```

Вывод сообщений об ошибке в консоль

```
$this->error('Что-то пошло не так!');
```

Взаимодействие с пользователем

Вы также можете воспользоваться методами `ask` и `confirm` для обеспечения пользовательского ввода:

Попросить пользователя ввести данные:

```
$name = $this->ask('Как Вас зовут?');
```

Попросить пользователя ввести секретные данные:

```
$password = $this->secret('Какой пароль?');
```

Попросить пользователя подтвердить что-то:

```
if ($this->confirm('Вы желаете продолжить? [yes|no]'))
{
    //
}
```

Вы таежке можете указать ответ по умолчанию для метода `confirm`. Это должно быть `true` или `false`:

```
$this->confirm($question, true);
```

Регистрация Команд

Регистрация команды Artisan'a

Как только Вша команда будет готова, Вам нужно зарегистрировать ее в Artisan'e, чтобы воспользоваться ею. Обычно это делается в файле `app/start/artisan.php`. В этом файле вы можете вопользоваться методом `Artisan::add` для того, чтобы зарегистрировать команду:

```
Artisan::add(new CustomCommand);
```

Регистрация команды, зарегистрированной в IoC контейнере

Если Ваша команда зарегистрирована в IoC контейнере приложения, необходимо воспользоваться методом `Artisan::resolve`, чтобы команда стала доступной Artisan'y:

```
Artisan::resolve('binding.name');
```

Регистрация команд внутри сервис-провайдеров (Service Provider)

Если Вам необходимо зарегистрировать команды внутри сервис-провайдера, следует вызывать метод `commands` из метода `boot` провайдера, передавая в качестве аргумента зарегистрированное имя в IoC контейнере:

```
public function boot()
{
    $this->commands('command.binding');
}
```

Вызов других команд

Иногда может потребоваться вызвать другую команду из Вашей команды. Это можно сделать, вызвав метод `call`:

```
$this->call('command:name', array('argument' => 'foo', '--option' => 'bar'));
```