Введение

Философия Laravel

Laravel - библиотека для построения вёб-приложений с выразительным, элегантным синтаксисом. Мы верим, что процесс разработки должен быть творческим и доставлять вам удовольствие. Laravel стремится устранить общеизвестные проблемы, возникающие при разработке интернет-приложений, облегчая работу с такими часто используемыми возможностями, как авторизация, маршрутизация, сессии и кэширование.

Задача Laravel - сделать разработку приятной для самого разработчика, в то же время не ущемляя функциональности его приложения. Счастливые разработчики пишут самый лучший код. Мы постарались совместить всё самое лучшее, что мы видели в других вёб-библиотеках, в том числе на других языках, таких как Ruby on Rails, ASP.Net MVC и Sinatra.

Laravel обладает широкими возможностями, но при этом прост для понимания и содержит инструменты, необходимые для больших и крепких приложений. Замечательный контейнер обратного управления (IoC), выразительная система миграций и тесно связанная система юнит-тестов даёт вам всё, что нужно для решения любой задачи, которая перед вами стоит.

Изучение Laravel

Один из лучших способов познакомиться с Laravel - прочитать все разделы документации. Это руководство рассматривает все компоненты фреймворка и как использовать их в вашем приложении.

Вы также можете посмотреть книги по Laravel. Они написаны сообществом и являются хорошими помощниками в изучении Laravel 4.

- Code Bright by Dayle Rees
- Laravel Testing Decoded by Jeffrey Way
- Laravel: From Apprentice To Artisan by Taylor Otwell
- Implementing Laravel by Chris Fidao
- Getting Stuff Done With Laravel 4 by Chuck Heintzelman

Команда разработчиков

Laravel был создан Taylor Otwell, который продолжает вести его разработку по сей день. Другие известные члены сообщества: Dayle Rees, Shawn McCool, Jeffrey Way, Jason Lewis, Ben Corlett, Franz Liedke, Dries Vints, Mior Muhammad Zaki и Phil Sturgeon.

Спонсоры Laravel

Следующие организации оказали финансовую поддержку разработке этой библиотеки:

- UserScape
- Cartalyst
- Elli Davis - Toronto Realtor
- Jay Banks - Vancouver Lofts & Condos
- Julie Kinnear - Toronto MLS
- Jamie Sarner - Toronto Real Estate

Быстрый старт

Установка

Чтобы установить Laravel вам нужно выполнить эту команду в командной строке:

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

Либо можно скачать архив хранилища с GitHub. Дальше, установите Composer, запустите `composer install` в корневой папке вашего проекта. Она загрузит и установит зависимости фреймворка.

После установки изучите структуру папок. Папка `app` одержит подпапки, такие как `views`, `controllers` и `models`. Большая часть кода вашего приложения будет находиться где-то внутри них. Вы также можете посмотреть на содержимое `app/config` и на настройки, которые доступны для изменения.

Маршрутизация

Для начала давайте создадим наш первый маршрут (route). В Laravel самый простой маршрут - функция-замыкание (Closure). Откройте файл `app/routes.php` и добавьте этот код в его конец:

```
Route::get('users', function()
{
    return 'Users!';
});
```

Теперь если вы перейдёте в браузере на адрес `/users` то должны увидеть текст `Users!`. Отлично! Вы только что создали свой первый маршрут.

Маршруты также могут быть привязаны к классу контроллера. Например:

```
Route::get('users', 'UserController@getIndex');
```

Этот маршрут сообщает Laravel, что запросы к `/users` должны вызывать метод `getIndex`

класса `UserController`. Для дополнительной информации см. раздел о контроллерах.

Создаём шаблон

Давайте теперь создадим простой шаблон, или вид (от англ. 'view'), чтобы показывать информацию о наших пользователях. Шаблоны находятся в `app/views` и содержат HTML-код вашего приложения. Мы создадим два новых шаблона в этой папке: `layout.blade.php` и `users.blade.php`. Начнём с `layout.blade.php`:

```
<html>
    <body>
        <h1>Laravel Quickstart</h1>

        @yield('content')
    </body>
</html>
```

Теперь создадим `users.blade.php`:

```
@extends('layout')

@section('content')
    Users!
@stop
```

Кое-что из этого кода, возможно, выглядит для вас весьма странно. Это из-за того, что мы используем шаблонизатор Laravel - Blade. Blade очень быстр благодаря тому, что это всего лишь набор регулярных выражений, которые преобразуют ваши шаблоны в чистый код на PHP. Blade позволяет наследовать шаблоны, а также добавляет "синтаксический сахар" к таким частоиспользуемым PHP-конструкциям, как `if` и `for`. Для инфромации см. раздел о Blade.

Теперь, когда у нас есть шаблоны, давайте используем их в нашем маршруте `/users`. Вместо возврата простой строки `Users!` мы вернём экземпляр шаблона:

```
Route::get('users', function()
{
    return View::make('users');
});
```

Замечательно! Вы создали шаблон маршрута, который наследует разметку страницы (шаблон layout). А теперь перейдём к работе с базой данных.

Создаём миграцию

Для создания таблицы для хранения наших данных мы используем систему миграций Laravel. Миграции позволяют вам определять изменения в БД, используя выразительный синтаксис, а затем легко делиться ими с остальными членами вашей команды.

Для начала настроим соединение с БД. Все соединения настраиваются в файле `app/config/database.php`. По умолчанию Laravel использует MySQL, и для работы вам нужно указать здесь параметры подключения. Если хотите, можете установить параметр `driver` в значение `sqlite` и Laravel будет использовать базу данных SQLite, которая находится в папке `app/database`.

Для создания миграции мы воспользуемся утилитой командной строки Artisan. Выполните следующую команду в корневой папке вашего проекта:

```
php artisan migrate:make create_users_table
```

Теперь найдите созданный файл миграции в папке `app/database/migrations`. Он содержит класс с двумя методами: `up` и `down`. В методе `up` вам нужно произвести требуемые изменения в таблицах, а в методе `down` вам нужно их откатить.

Давайте создадим такую миграцию:

```
public function up()
{
    Schema::create('users', function($table)
    {
        $table->increments('id');
        $table->string('email')->unique();
        $table->string('name');
        $table->timestamps();
    });
}

public function down()
{
    Schema::drop('users');
}
```

Теперь мы можем применить миграцию через командную строку, используя команду `migrate`. Просто выполните её в корне проекта:

```
php artisan migrate
```

Если вам нужно откатить миграцию - выполните команду `migrate:rollback`. Теперь, когда у нас есть таблица, начнём загружать данные!

Eloquent ORM

Laravel поставляется с замечательной ORM (механизм связывания записей БД с объектами PHP - прим. пер.) - Eloquent. Если вы программировали в библиотеке Ruby on Rails, то она покажется вам знакомой, так как Eloquent следует принципам ActiveRecord при взаимодействии с базами данных.

Для начала создадим модель. В Eloquent модель используется для запросов к соответствующей таблице в БД, а также представляет отдельную запись (record) внутри неё. Не беспокойтесь, скоро это станет куда понятнее! Модели обычно хранятся в папке `app/models`. Создадим файл `User.php` с таким кодом:

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

Заметьте, что нам не нужно указывать, какую таблицу нужно использовать. Eloquent следует множеству соглашений, одно из которых - то, что имя таблицы соответствует множественному числу имени класса её модели (user -> users - прим. пер.). Это очень удобно!

Добавьте новые записи в таблицу `users`, используя ваш любимый инструмент для работы с БД, и мы посмотрим, как Eloquent позволяет их получить, а затем передать в шаблон.

Итак, изменим наш маршрут `/users`:

```
Route::get('users', function()
{
    $users = User::all();

    return View::make('users')->with('users', $users);
});
```

Посмотрим, что здесь происходит. Сперва мы получаем все записи в таблице 'users' через метод 'all' модели 'User'. Дальше мы передаём эти записи шаблону через его метод 'with'. Этот метод принимает имя переменной и её значение и таким образом делает данные доступными внутри своего кода.

Отлично. Теперь мы готовы к тому, чтобы показать пользователей в нашем шаблоне!

Отображаем данные

Теперь, когда мы сделали переменную 'users' доступной для нашего шаблона мы можем отобразить её таким образом:

```
@extends('layout')
```

```
@section('content')
    @foreach($users as $user)
        <p>{{ $user->name }}</p>
    @endforeach
@stop
```

> **Примечание:** Код выше открыт для XSS-атак. Blade, как и простой код на PHP, не экранирует вывод, поэтому вам нужно следить, чтобы выводимые строки содержали экранированый HTML. Альтернативные шаблонизаторы, такие как `HTMLki`, делают это автоматически. - прим. пер.

Вам может быть интересно, куда подевались вызовы `echo`. Blade позволяет вам выводить строки, обрамляя их двойными фигурными скобками ({{ ... }}). Теперь вы можете перейти в браузере к своему маршруту и увидеть имена всех имеющихся пользователей.

Это только начало. В этом руководстве вы ознакомились с самыми основами Laravel, но у него есть ещё очень много интересных вещей, которые вам стоит узнать. Продолжайте читать документацию и глубже узнавать возможности, предоставляемые Eloquent и Blade. А может вам больше интересны очереди и юнит-тесты. Или же вам хочется размять мускулы с контейнером IoC. Выбор за вами!

Помощь проекту

Введение

Laravel - бесплатный проект с открытым исходным кодом. Это значит, что любой может помочь в его разработке и развитии. На текущий момент исходный код Laravel расположен на Github, что даёт возможность легко создавать ваши ветки и отправлять изменения в главное хранилище.

Запросы на слияние

Перед отправкой запроса на слияние (pull request) для новой возможности вы должны сначала создать запись в багтрекере с надписью `[Proposal]` в заголовке. Она будет просмотрена и либо принята, либо отклонена. Как только предложене принято, к записи можно прикрепить новый запрос на слияние. Запросы, не соответствующие правилам, будут сразу закрываться.

Отправка кода с исправлениями может происходить без создания записи. Если вы уверены, что знаете решение проблемы, упомянутой на Github, оставьте комментарий с подробностями о предлагаемом изменении.

Дополнения и исправления документации могут вноситься через это хранилище Github.

Запросы возможностей

Если у вас есть идея, которую вы хотите видеть воплощённой в Laravel, вы можете создать запись в багтрекере на Github с надписью `[Request]` в заголовке. Запрос будет оценён одним из членов команды проекта.

Стандарты кода

Laravel следует стандартам PSR-0 и PSR-1. В дополнение к ним есть следующие правила, которые должны выполняться:

- Объявления пространств имён должны писаться на той же линии, что и открывающий тег `<?php`.
- Открывающая фигурная скобка `{` для классов должна писаться на той же строке, что и имя класса.
- Открывающая фигурная скобка `{` для функций и блочных операторов должна писаться на

отдельной строке.
- К именам интерфейсов добавляется слово `Interface` (`FooInterface`).

Installation

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

Создание проекта Composer

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

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

Загрузка архива

Как только Composer установлен скачайте последнюю версию фреймворка и извлеките архив в папку на вашем сервере. Дальше, в корне вашего приложения на Laravel выполните `php composer.phar install` (или `composer install`) для установки всех зависимостей библиотеки. Этот процесс требует, чтобы на сервере 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.

> **Примечание:** В Laravel 3 и в ранних версиях Laravel 4 единственная настройка, которую вам нужно было изменить - `key` в файле `app/config/app.php`. Это значение должно быть случайной строкой длиной 32 символа. Оно используется при шифровании и зашифрованные строки не будут безопасными, пока вы не измените эту настройку. Теперь в Laravel 4 это делается автоматически. Вы также можете быстро его установить с помощью следующей команды: `php artisan key:generate`.

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`. Рекомендуется, чтобы вы установили эту папку корневой папкой вашего сайта (DocumentRoot в Apache) или переместили её содержимое в корневую папку сайта, а все другие файлы фреймворка - за её пределы.

Красивые URL

Laravel поставляется вместе с файлом `public/.htaccess`, который настроен для обработки URL без указания `index.php`. Если вы используете Apache в качестве вёб-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]
```

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

Часто необходимо иметь разные значения для разных настроек в зависимости от среды, в которой выполняется приложение. Например, вы можете захотеть использовать разные драйвера кэша на локальном и производственном 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'),

));
```

Вы можете передать в метод функцию-замыкание (Closure), что позволит вам делать собственные проверки:

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

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

Получение имени текущей среды:

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

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. Вызов `App::down` 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);
});
```

Прохождение запроса

Введение

В Laravel, жизненный цикл запроса весьма прост. Запрос поступает в ваше приложение, где он направляется к определённому маршруту или контроллеру. Полученный результат отправляется обратно к клиенту и отображается в его браузере. Иногда вам может потребоваться сделать какие-то действия перед или после вызова маршрута. Для этого есть несколько путей, два из которых - старт-файлы и события.

Старт-файлы

Файлы запуска, или 'старт-файлы' вашего приложения хранятся в папке `app/start`. По умолчанию создано три таких файла: `global.php`, `local.php`, и `artisan.php`. Для информации об `artisan.php`, см. соответствующий раздел.

Файл `global.php` изначально содержит несколько начальных вызовов, таких как регистрация журнала (Logger) и подключение `app/filters.php`. Однако вы можете добавить сюда всё, что вам нужно. Этот файл автоматически выполняется при *каждом* запросе вне зависимости от среды, в которой выполняется ваше приложение. Файл `local.php` вызывается только, когда приложение работает в среде `local`. Больше информации о средах можно найти в разделе о настройках.

Конечно, если у вас определены другие среды, кроме `local`, вы можете создать старт-файлы для любой из них. Они будут автоматически загружаться, когда приложение работает в соответствующей среде.

События

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

Регистрация обработчиков событий

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

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

Эти обработчики будут запущены соответственно `до` и `после` каждого вызова в вашем приложении.

Маршрутизация

Простейшая маршрутизация

Большинство маршрутов (routes) вашего приложения будут определены в файле `app/routes.php`. В Laravel, простейший маршрут состоит из URI (пути) и функции-замыкания.

Простейший GET-маршрут:

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

Простейший POST-маршрут:

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

Регистрация маршрута для любого типа HTTP-запроса:

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

Регистрация маршрута, всегда работающего через HTTPS:

```
Route::get('foo', array('https', function()
{
```

```
    return 'Must be over HTTPS';
}));
```

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

```
$url = URL::to('foo');
```

Параметры маршрутов

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

Необязательные параметры маршрута:

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

Необязательные параметры со значением по умолчанию:

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

Маршруты с соответствием пути регулярному выражению:

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

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]+'))
```

Фильтры маршрутов

Фильтры - удобный механизм ограничения доступа к определённому маршруту, 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 на сам запрос и обработчик маршрута не будет вызван, и все `after`-фильтры тоже будут пропущены.

Привязка фильтра к маршруту:*

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

Привязка нескольких фильтров к маршруту:

```
Route::get('user', array('before' => '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, $value)
{
    //
});
```

Фильтры по шаблону

Вы можете также указать, что фильтр применяется ко всем маршрутам, 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 замыканий. Так как фильтры создаются с помощью IoC-контейнера, то вы можете положиться на его внедрение зависимостей для лучшего тестирования.

Определение класса для фильтра:

```
class FooFilter {

    public function filter()
```

```
    {
        // Filter logic...
    }

}
```

Регистрация фильтра-класса:

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

Именованные маршруты

Присваивая имена маршрутам вы можете сделать обращение к ним (при генерации URL или переадресациях) более удобным. Вы можете задать имя маршруту 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 маршрутов

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 привязан фильтр auth.
    });

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

Доменная маршрутизация

Маршруты Laravel способны работать и с поддоменами по их маске и передавать в ваш обработчик параметры из шаблона.

Регистрация маршрута по поддомену:

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

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

});
```

Префикс пути

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

Добавление префикса к сгруппированым маршрутам:

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

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

});
```

Привязка моделей

Привязка моделей - удобный способ передачи экземпляров моделей в ваш маршрут. 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 возбуждено исключение 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 NotFoundException;
});
```

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 - возбудив 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.

Маршрутизация 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 ввод

Базовый ввод

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');
```

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 клиент изменит их значение, то они станут неверными.

Чтение cookie:

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

Добавление cookie к ответу:

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

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

Создание cookie, которая хранится вечно:

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

Старый ввод

Вам может пригодиться сохранение пользовательского ввода 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 ввод 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 старого ввода:

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

Файлы

Получение объекта загруженного файла:

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

Определение успешной загрузки:

```
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.

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

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();
```

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');
```

Сделан ли запрос через AJAX:

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

Использует ли запрос HTTPS:

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

Шаблоны и отклики

Базовые отклики

Возврат строк из маршрутов:

```
Route::get('/', function()
{
    return 'Привет, мир!';
});
```

Создание собственного ответа (отклика)

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

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

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

return $response;
```

Добавление cookie к ответу:

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

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

Переадресация

Returning A Redirect

```
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 маршрут:

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

Переадресация на именованный маршрут с параметрами:

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

Переадресация на именованный маршрут с именованными параметрами:

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

Переадресация на действие контроллера

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

Переадресация на действие контроллера с параметрами:

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

Переадресация на действие контроллера с именованными параметрами:

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

Шаблоны

Шаблоны (views) обычно содержат HTML-код вашего приложения и представляют собой удобный способ разделения логики контроллеров и обработки страниц от их представления. Шаблоны хранятся в папке `app/views`.

Простой шаблон может иметь такой вид:

```
<!-- Шаблон из файла app/views/greeting.php -->
```

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

Этот шаблон может быть возвращён в браузер таким образом:

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

Второй параметр, переданный в `View::make` - массив данных, которые должны быть доступны внутри шаблона.

Передача переменных в шаблон:

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

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

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

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

Вы также можете "поделить" данные между всеми шаблонами, сделав их глобальной:

```
View::share('name', 'Стив');
```

Передача вложенного шаблона в шаблон

Иногда вам может быть нужно передать шаблон внутрь другого шаблона. Например, имея шаблон `app/views/child/view.php` нам может понадобиться передать его в другой шаблон и мы можем сделать это так:

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

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

Затем вложенный шаблон может быть отображён в родительском шаблоне:

```
<html>
    <body>
```

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

Составители

Составители шаблонов - функции обратного вызова или методы класса, которые вызываются, когда данный шаблон формируется (отображается) в строку. Если у вас есть данные, которые вы хотите привязать к шаблону при каждом его формировании, то составители помогут вам выделить такую логику в единое место. Таким образом, составители могут работать как "модели шаблонов" или "представители".

Регистрация составителя:

```
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 составитель сразу к неским шаблонам:

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

Если вам больше нравятся методы классов, что позволяет вам использовать регистрации в IoC Container, то вы можете сделать это так:

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

Класс составителя должен иметь такой вид:

```
class ProfileComposer {

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

}
```

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

Создатели

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

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

Особые отклики

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 отклика загрузки файла:

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

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

Контроллеры

Простейшие контроллеры

Вместо того, чтобы определять всю маршрутизацию (routing) вашего проекта в файле `routes.php` вы можете организовать её, 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`.

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 маршрут для его действия (action):

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

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

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

> **Примечание:** Помните, что имена класса в строках следуют обычным правилам PHP и если ваш класс начинается с `n`, `t` и других букв в нижнем регистре, то обратный слэш перед ними нужно экранировать - иначе они преобразуются в разрыв строки, табуляцию или иной спецсимвол: `namespace\new_controller` - прим. пер.

Вы также можете присвоить имя этому маршруту:

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

Вы можете получить URL к действию методом `URL::action` method:

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

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

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

Фильтры для контроллеров

Фильтры могут указываться для контроллеров аналогично "обычным" маршрутам:

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

Однако вы можете указывать их и изнутри 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()
        {
            //
        });
    }

}
```

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`:

Регистрация RESTful-контроллера:

```
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()
    {
        //
```

```
    }

}
```

Методы `index` обрабатывают корневой URI контроллера - в нашем случае это `users`.

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

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

Контроллеры ресурсов

Они 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', 'delete')));
```

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.

Defining A Catch-All Method

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

Ошибки и журнал

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`. **Настоятельно рекомендуется отключать детализацю ошибок для производственных серверов.**

Обработка ошибок

Файл `app/start/global.php` по умолчанию содержит обработчик любых 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)
{
    //
});
```

Исключения HTTP

Такие исключения могут возникнуть во время обработки запроса от клиента. Это может быть ошибка "Не найдено" (404), "Требуется авторизация" (401) или даже "Ошибка сервера" (500). Для того, чтобы отправить такой ответ, используйте следующее:

```
App::abort(404, 'Страница не найдена.');
```

Первый параметр - код HTTP-ответа, а второй - сообщение, которое вы хотите показать вместе с ошибкой.

В случае с "Требуется авторизация" (401) сделайте аналогичный вызов:

```
App::abort(401, 'Вы не вошли в систему.');
```

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);
});
```

Журнал

Стандартный механизм журналирования представляет собой простую надстройку над мощной системой Monolog.По умолчанию Laravel настроен для создания одного файла журнала на один день и хранения их в `app/storage/logs`. Вы можете записывать в них 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 критичнорсти, 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 события для отслеживания всех новых сообщений.

Отслеживание новых сообщений в журнале:

```
Log::listen(function($level, $message, $context)
{
    //
});
```

Кэш

Настройка

Laravel предоставляет унифицированное API для различных систем кэширования. Настройки кэша содержатся в файле `app/config/cache.php`.Здесь вы можете указать драйвер, используемый по умолчанию внутри вашего приложения. Laravel изначально 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);
```

Запись элемента, если он не существует:

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

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 упаковываются (сериализуются), поэтому вы можете хранить любые типы.

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);
```

Группы элементов

> **Внимание:** группы не поддерживаются драйверами `file` и `database`.

Вы можете объединять элементы кэша в группы (секции), а затем сбрасывать всю группу целиком. Для доступа к группе используйте метод `section`.

Обращение к группе элементов кэша:

```
Cache::section('people')->put('John', $john);

Cache::section('people')->put('Anne', $anne);
```

Вы можете использовать обычные операции над элементами группы, такие как чтение, запись, `increment` и `decrement`.

Чтение элемента группы:

```
$anne = Cache::section('people')->get('Anne');
```

Вы также можете удалить всю группу:

```
Cache::section('people')->flush();
```

Кэширование в базе данных

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

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

```
});
```

События

Простейшее использование

Класс `Event`содержит простую реализацию концепции "Наблюдатель", что позволяет вам подписываться на уведомления о событиях в вашем приложении.

Подписка на событие:

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

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

Возбуждение события:

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

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.

Подписка на событие с приоритетом:

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

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

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

Прерывание обработки события:

```
Event::listen('user.login', function($event)
{
```

```
    // Обработка события...

    return false;
});
```

Обработчики по шаблону

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

Регистрация обработчика по шаблону

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

Этот обработчик будет вызываться при любом событии, начинающемся `foo.`. Обратите внимание, что в качестве первого аргумента ему передаётся полное имя обрабатываемого события.

Классы-обработчики

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

Регистрация обработчика в классе:

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

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

Создание обработчика внутри класса:

```
class LoginHandler {

    public function handle($data)
    {
        //
    }

}
```

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

Регистрация обработчика в другом методе:

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

Запланированные события

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

Регистрация цепочки событий:

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

Регистрация "пускателя":

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

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

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

Классы-подписчики

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

Определение класса-подписчика:

```
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('user.login', 'UserEventHandler@onUserLogin');

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

}
```

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

Регистрация класса-подписчика:

```
$subscriber = new UserEventHandler;

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

Фасады

Введение

Фасады предоставляют "статический" интерфейс к классам, доступным в контейнере IoC. 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.

> **Внимание:** перед погружением в фасады настоятельно рекомендуется как можно детальнее изучить контейнер IoC.

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.

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 имя привязки (типа) 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. Связка 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()
    {
        //
    }

}
```

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 поставщики поставщик услуг 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 псевдоним (alias) 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 псевдонимов

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

Фасады-заглушки

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 фасады-заглушки соответствующем разделе документации.

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'))
```

> **Внимание:** 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

Laravel предоставляет простую защиту от подделки межсайтовых запросов. Сперва случайная последовательность символов 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`.

Добавление CRSF-строки в форму:

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

Присоединение CRSF-фильтра к маршруту:

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

Привязка модели к форме

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

Открытие формы для модели:

```
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. 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.

> **Внимание:** 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'));
```

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

Текстовые и скрытые поля

**Создание текствого поля ввода:*

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

Указание значения по умолчанию:

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

> **Внимание:** 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());
```

Флажки и кнопки переключения

Генерация флажка или кнопки переключения (radio button):

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

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

Генерация флажка или кнопки, 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');
```

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!');
```

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

Макросы

К классу Form можно легко добавлять собственные методы; они называются "макросами". Вот как это работает. Сперва зарегистрируйте макрос с нужным именем и функцией-замыканием.

Регистрация макроса для Form:

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

Теперь вы можете вызвать макрос, используя его имя.

Вызов макроса:

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

Функции

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 плоским, 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('name' => 'Taylor'), array('name' => 'Dayle'));

var_dump(array_fetch($array, 'name'));

// array('Taylor', 'Dayle');
```

array_first

Вернуть первый элемент массива, прошедший 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')));

$array = array_forget($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');
```

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'];
}));
```

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
```

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).

```
$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).

```
$singular = str_singular('cars');
```

studly_case

Преобразовать строку в `StudlyCase`.

```
$value = studly_case('foo_bar');

// FooBar
```

trans

Перевести переданную языковую строку. Псевдоним для `Lang::get`.

```
$value = trans('validation.required'):
```

trans_choice

Перевести переданную языковую строку с изменениями. Псевдоним для `Lang::choice`.

```
$value = trans_choice('foo.bar', $count);
```

URLs

action

Сгенерировать URL для заданного действия контроллера.

```
$url = action('HomeController@getIndex', $params);
```

route

Сгенерировать URL для заданного именованного маршрута.

```
$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-последовательности.

```
$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();
```

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 и мощных приложений, так же как и для внесения изменений в код ядра самого фреймворка.

Основы использования

Есть два способа, которыми IoC-контейнер разрешает зависимости: через функцию-замыкание и через автоматическое определение. Для начала мы исследуем замыкания. Первым делом некий "тип" должен быть помещён в контейнер.

Помещение типа в контейнер:

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

Извлечение типа из контейнера:

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

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

Иногда вам может понадобиться поместить в контейнер тип, котоый должно быть извлечён (создан) только один раз, и чтобы все последующие вызовы возвращали бы тот же объект.

Помещение "разделяемого" типа в контейнер:

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

Singleton - шаблон проектирования Одиночка - прим. пер.

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

Помещение готового экземпляра в контейнер:

```
$foo = new Foo;

App::instance('foo', $foo);
```

Автоопределение класса

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

Автоопределение класса

```
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 рефлексии PHP для изучения класса и чтения подсказок типов (type hints) в его конструкторе. С помощью этой информации контейнер сам создаёт экземпляр класса.

Однако 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.

Связывание интерфейса и реализации:

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

Теперь посмотрим на следующий ((controllers 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 стабильности вашего приложения. Основной пример - зависимости при использовании контроллеров. Все контроллеры извлекаются из IoC, что позволяет вам использовать зависимости на основе подсказок типов в их конструкторах - ведь они будут определены автоматически.

Подсказки типов для указания зависимостей контроллера:

```
class OrderController extends BaseController {

    // OrderRepository - подсказка типа - имя класса передаваемого
аргумента-объекта.
    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 для `OrderRepository` может быть добавлен в контейнер, таким образом легко имитируя взаимодействие с БД.

Фильтры, составители, and классы-обработчики могут также извлекаться из IoC. При регистрации этих объектов просто передайте имя класса, который должен быть использован.

Другие примеры использования IoC:

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

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

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

Поставщики услуг

Поставщики услуг (service providers) - 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 команду (Artisan).

На самом деле бОльшая часть компонентов Laravel включает поставщиков услуг. Все зарегистрированные поставщики в вашем приложении указаны в массиве `providers` файла настроек `app/config/app.php`.

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

Создание поставщика услуг:

```
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 поставщика и готовы зарегистрировать его в своём приложении просто добавьте его в массиве `providers` файла настроек `app`.

Кроме этого, вы можете зарегистрировать его "на лету", используя метод `App::register`:

Регистрация поставщика услуг во время выполнения:

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

События контейнера

Контейнер IoC возбуждает событие каждый раз при извлечении объекта. Вы можете отслеживать его с помощью метода `resolving`:

Регистрация обработчика события:

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

Созданный объект передаётся в функцию в виде первого 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
```

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

Пример языкового файла:

```
<?php

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

Язык по умолчанию указан в файле настроек `app/config/app.php`. Вы можете изменить текущий язык во время работы вашего приложения методом `App::setLocale`:

Изменение языка по умолчанию "на лету":

```
App::setLocale('es');
```

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.

> **Внимание**: 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'))
{
    //
}
```

Множественное число

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);
```

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',
```

Работа с e-mail

Настройка

Laravel предоставляет простой интерфейс к популярной библиотеке SwiftMailer. Главный файл настроек - `app/config/mail.php`он 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 стандуртную 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 - `$data`, массив переменных, передаваемых в шаблон. Третий - функция-замыкание, позволяющая вам настроить новое сообщения.

> **Внимание:** переменная `$message` всегда передаётся в ваш шаблон и позволяет вам прикреплять вложения. Таким образом, вам не стоит передавать одноимённую переменную в параметре `$data`.

В дополнение к телу сообщения в формате HTML вы можете указать текстовое представление:

```
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));
```

> **Внимание:** Объект, передаваемый замыканию метода `Mail::send`, наследует класс собщения SwiftMailer, что позволяет вам вызывать любые методы для создания своего сообщения.

Добавление строчных вложений

Обычно добавление строчных вложений обычно утомительное занятие, однако Laravel делает его проще, позволяя вам добавлять файлы и получать соответствующие CID.

> Строчные 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 файла из потока данных:

```
<body>
    А вот картинка, полученная из строки с данными:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

Переменная `$message` всегда передаётся шаблонам сообщений классом `Mail`.

Очереди отправки

Из-за того, что отправка сообщений может сильно повлиять на время обработки запроса 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()`:

Помещение сообщения в очередь отправки:

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

Вы можете задержать отправку сообщения на нужное число секунд методом `later`:

```
Mail::later(5, 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'Джон Смит')->subject('Привет!');
});
```

Если же вы хотите поместить сообщение в определённую очередь отправки, то используйте методы `queueOn` и `laterOn`:

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

Локальная разработка

При разработке приложения обычно предпочтительно отключить доставку отправляемых сообщений. Для этого вы можете либо вызывать метод `Mail::pretend`, либо установить параметр `pretend` в значение `true` в файле настроек `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 любой библиотеки, а не только Laravel. Примерами таких отдельных пакетов являются Carbon и Behat. Любая из них может быть использована в Laravel после простого их указания 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 маршруты, 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` интерфейса 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, Тейлор Отуелл (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 `ServiceProvider` - поставщика услуг, который был создан для нового пакета. 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.

> **Внимание:** если ваш поставщик услуг 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, языковые строки, миграции и шаблоны - прим. пер.). Пакеты, как и обычные приложения, могут содержать любой из этих ресурсов.

Поставщики услуг

Поставщик услуг - просто начальный загрузчик для пакета. По умолчанию они могут содержать два метода: `boot` и `register`. Внутри этих методов вы можете выполнять любой код - подключать файл с маршрутами, регистрировать связи в контейнере IoC, устанавливать обработчики событий или что-то ещё.

Метод `register` вызывается сразу после регистрации поставщика услуг, тогда как команда `boot` вызывается только перед тем, как будет обработан запрос. Таким образом, если вашему поставщику требуется другой поставщик, который уже был зарегистрирован, или вы перекрываете услуги, зарегистрированные другим поставщиком - 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.

Для поставщиков услуг не существует "места по умолчанию". Вы можете поместить их в любое место, возможно, сгруппировав в пространство имён `Providers` в папке `app`. Этот файл может располагаться где угодно - главное, чтобы Composer мог загрузить его с помощью auto-loading facilities.

Соглашения

При использовании ресурсов из пакета, таких как настройки или шаблоны, то для отделения имени пакета обычно используют двойное двоеточие (::).

Загрузка шаблона из пакета:

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

Чтение параметров настройки в пакете:

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

> **Внимание:** если ваш пакет содержит миграции, подумайте о том, чтобы её имя с имени пакета для предотвращения возможных конфликтов с именами классов в других пакетах.

Процесс разработки

При разработке пакета бывает удобно работать в контексте вашего приложения, просматривая и экспериментируя с шаблонами и пр. Для начала сделайте чистую установку Laravel, затем используйте команду `workbench` для создания структуры пакета.

После того, как пакет создан вы можете сделать `workbench` изнутри папки `workbench/[vendor]/[package]`, а затем - `git push` для отправки пакета напрямую в хранилище. 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 маршрутами просто подключите его через `include` из метода `boot` вашего поставщика услуг.

Подключение файла с маршрутами из поставщика услуг:

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

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

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

Настройки пакетов

Некоторые пакеты могут требовать файлов настройки. 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 (::).

Чтение настроек пакета:

```
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.

Чтение настроек пакета:

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

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`.

Ручная регистрация пространства имён ресурсов:

```
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`.

Перекрытие файлов настроек

Когда другие разработчики устанавливают ваш пакет им может потребоваться перекрыть 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.

Выполнение команды опубликования настроек:

```
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.

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

Миграции пакетов

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

Создание миграций для пакета в `workbench`:

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

Выполнение миграций пакета в `workbench`:

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

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

Выполнение миграций установленного пакета:

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

Внешние ресурсы пакетов

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`, поэтому нам нужно перенести их в папку `public` нашего приложения. Команда `asset:publish` выполнит это за вас.

Перемещение ресурсов пакета в папку `public`:

```
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 коде шаблонов вашего пакета.

Публикация пакетов

Когда ваш пакет готов к опубликованию, вам нужно отправить его в хранилище 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.

Страничный вывод

Настройка

В других фрейморках страничный вывод может быть большой проблемой. Laravel же делает этот процесс безболезненным. В файле настроек `app/config/view.php` есть единственный параметр: `pagination` - который указывает, какой (views шаблон) 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 изначально совместимы с Twitter Bootstrap.**

Использование

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

Страничный вывод выборки из БД:

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

Страничный вывод запроса Eloquent:*

```
$users = User::where('votes', '>', 100)->paginate(15);
```

Аргумент, передаваемый методу `paginate` - число строк, которые вы хотите видеть на одной странице. Как только вы получили результаты вы можете показать их с помощью шаблона и создать ссылки на страницы 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.

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

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

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

Создание страничного вывода вручную:

```
$paginator = Paginator::make($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

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`

Основы использования

Добавление новой задачи в очередь:

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

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

Регистрация обработчика задачи:

```
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.

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

Задача с произвольным методом-обработчиком:

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

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

Удаление выполненной задачи:

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

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

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

Помещение задачи обратно в очередь:

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

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

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

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

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

Получение числа попыток запуска задания:

```
if ($job->attempts() > 3)
```

```
{
    //
}
```

Получение идентификатора задачи:

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

Добавление замыканий

Вы можете помещать в очередь и функции-замыкания. Это очень удобно для простых, быстрых задач, выполняющихся в очереди.

Добавление замыкания в очередь:

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

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

> **Внимание:** константы `__DIR__` и `__FILE__` не должны использоваться в замыканиях.

При использовании push-очередей Iron.io, будьте особенно внимательны при добавлении замыканий. Конечная точка выполнения, получающая ваше сообщение, 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.

Работа сервера приёма очереди

Laravel включает в себя задание Artisan, которое будет выполнять новые задачи по мере их поступления. Вы можете запустить его командой `queue:listen`:

Запуск сервера приёма:

```
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.

Кроме этого, вы можете указать число секунд, в течении которых будут выполняться задачи.

Указание числа секунд для работы сервера:

```
php artisan queue:listen --timeout=60
```

Для обработки только первой задачи можно использовать команду `queue:work`.

Обработка только первой задачи в очереди:

```
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`.

После этого вы можете использовать команду `queue:subscribe` Artisan для регистрации URL конечной точки, которая будет получать добавляемые в очередь задачи.

Регистрация подписчика push-очереди:

```
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 конечной 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.

Безопасность

Настройка

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

По умолчанию Laravel включает в себя модель `User` в папке `app/models`, которая может использоваться вместе с драйвером авторизации Eloquent (по умолчанию). При создании таблицы для данной модели убедитесь, что поле пароля принимает как минимум 0 символов.

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

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');
}
```

Авторизация пользователей

Для авторизации пользователя в вашем приложении используется 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 авторизации. Дополнительный URL может быть передан в метод, если требуемый адрес не доступен.

Когда вызывается метод `attempt` возбуждается событие `auth.attempt`. При успешной авторизации также произойдёт событие `auth.login`.

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

Проверка авторизации пользователя:

```
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).

Авторизация и запоминание пользователя:

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

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

Вы также можете передать дополнительные условия для запроса к таблице.

Авторизация пользователя с условиями:

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

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

Доступ к авторизованному пользователю:

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

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

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

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

Проверка данных для входа без авторизации:

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

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

Авторизация пользоваля на один запрос:

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

Выход пользователя из системы:

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

Ручная авторизация

Если вам нужно авторизовать существующего пользователя просто передайте его модель в метод `login`:

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

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

Это эквивалентно авторизации пользвоателя через его данные методом `attempt`.

Защита маршрутов

Вы можете использовать Фильтры маршрутов, чтобы позволишь только авторизованным пользователям обращаться к данному маршруту. Изначально Laravel содержит фильтр `auth`, который содержится в файле `app/filters.php`.

Защита маршрута:

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

Защита от подделки запросов (CSRF)

Laravel предоставляет простой способ защиты вашего приложения от подделки межсайтовых запросов 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!';
}));
```

Простая HTTP-авторизация

HTTP Basic Authentication - простой и быстрый способ авторизовать пользователей вашего приложения без создания дополнительной страницы входа. Для начала подключите фильтр `auth.basic`.

Защита маршрута фильтром HTTP-авторизации:

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

По умолчанию, фильтр `basic` будет использовать поле `email` модели объекта при авторизации. Если вы хотите использовать иное поле, можно передать его имя первым параметром методу `basic`:

```
return Auth::basic('username');
```

Вы можете использовать HTTP-авторизацию без установки cookies в сессии, что особенно удобно для авторизации в API. Для того, чтобы сделать это, зарегистрируйте фильтр, вовращающий результат вызова `onceBasic`.

Авторизация без запоминание состояния:

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

Сброс и изменение паролей

Отправка письма о сбросе

Большая часть приложений позволяют пользователям сбрасывать свои забытые пароли. Вместо того, чтобы заставлять вам реализовывать эту функциональность в каждом приложении, Laravel предоставляет удобный способ для отправки писем о сбросе пароля и выполнения самого сброса. Для начала убедитесь, что модель `User` реализует интерфейс `Illuminate\Auth\Reminders\RemindableInterface`. Это уже сделано для `User`, включённой в фреймворк по умолчанию.

Реализация интерфейса `RemindableInterface`:

```
class User extends Eloquent implements RemindableInterface {

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

}
```

Затем вам нужно создать таблицу для хранения ключей сброса паролей. Для создания миграции для таблицы просто выполните команду `auth:reminders` интерфейса Artisan.

Генерация миграции для таблиц сброса паролей:

```
php artisan auth:reminders

php artisan migrate
```

Для отправки уведомления вы можете использовать метод `Password::remind`.

Отправка уведомления о сбросе пароля:

```
Route::post('password/remind', function()
{
    $credentials = array('email' => Input::get('email'));

    return Password::remind($credentials);
});
```

Заметьте, что параметры, переданные в метод `remind`, похожи на те, которые передаются методу `Auth::attempt`. Этот метод получит модель `User` и отправит соответствующее письмо со ссылкой для сброса. В шаблон письма будет передана переменная `token`, которая может быть использвоана для изменения пароля через форму его сброса. Кроме неё будет также передана переменная `user`.

> **Внимание:** вы можете указать, какой шаблон должен использоваться при создании сообщения, изменив настройку приложения `auth.reminder.email`. Изначально фреймворк уже содержит нужный шаблон.

Вы можете изменить объект письма, которое отправляется пользователю, передав замыкание в виде второго аргумента методу `remind`:

```
return Password::remind($credentials, function($message, $user)
{
    $message->subject('Ваше уведомление о сбросе.');
```

```
});
```

Как вы можете заметить, в маршруте мы напрямую возвращаем результат вызова метода `remind`. По умолчанию этот метод возвращает переадресацию на текущий адрес, если возникла ошибка при сбросе пароля, при этом устанавливается одноразовая переменная `error`, а также `reason`, которая используется для извлечения языковой строки из языкового файла `reminders`. Если пароль был успешно сброшен, то будет установлена одноразовая переменная `success`. Таким образом, шаблон для формы сброса пароля должен выглядеть примерно так:

```
@if (Session::has('error'))
    {{ trans(Session::get('reason')) }}
@elseif (Session::has('success'))
    На ваш e-mail было отправлено письмо с инструкциями о сбросе пароля.
@endif

<input type="text" name="email">
<input type="submit" value="Отправить письмо">
```

Сброс пароля

Как только пользователь перешёл по ссылке в письме, он будет перенаправлен на форму со скрытым ключом `token`, а также полями `password` и `password_confirmation`. Ниже - пример маршрута для формы сброса:

```
Route::get('password/reset/{token}', function($token)
{
    return View::make('auth.reset')->with('token', $token);
});
```

А сама форма может выглядеть так:

```
@if (Session::has('error'))
    {{ trans(Session::get('reason')) }}
@endif

<input type="hidden" name="token" value="{{ $token }}">
<input type="text" name="email">
<input type="password" name="password">
<input type="password" name="password_confirmation">
```

Обратите внимание, что мы снова используем `Session` для отображения ошибки, которая могла произойти при сбросе пароля. Дальше мы определяем POST-маршрут, который и произведёт сброс:

```
Route::post('password/reset/{token}', function()
```

```
{
    $credentials = array('email' => Input::get('email'));

    return Password::reset($credentials, function($user, $password)
    {
        $user->password = Hash::make($password);

        $user->save();

        return Redirect::to('home');
    });
});
```

При успешном сбросе объект `User` и пароль будут переданы в замыкание, что позволит вам сохранить изменённую модель. После этого вы можете вернуть объект `Redirect` или любой другой тип ответа, что и будет возвращено методом `reset`. Заметьте, что этот метод автоматически проверяет переданный ключ `token`, данные для входа и совпадение введённых паролей.

Также, по аналогии с методом `remind`, если во время сброса произошла ошибка метод `reset` вернёт объект `Redirect` на текущий адрес с одноразовыми переменными `error` и `reason`.

Шифрование

Laravel предоставляет функции для устойчивого шифрование по алгоритму AES-256 с помощью расширения mcrypt для PHP.

Шифрование строки:

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

> **Внимание:** обязательно установите 32-значный ключ `key` в файле `app/config/app.php`. Если этого не сделать, зашифрованные строки не будут безопасными.

Расшифровка строки:

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

You may also set the cipher and mode used by the encrypter:

Изменение алгоритма и режима шифрования:

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

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

Сессии

Настройка

HTTP-приложения не имеют состояний. Сессии - способ сохранения информации о клиенте между отдельными 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.

Основы использования

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 'умолчание'; });
```

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 данные

Иногда вам нужно сохранить переменную только для следующего запроса. 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 постовляется 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 эти быстрые кэширующие хранилища пар ключ/значение.
- `array` - данные содержатся в виде простых массивов PHP и не будут сохраняться между запросами.

> **Внимание:** драйвер `array` обычно используется для юнит-тестов, так как он на самом деле не сохраняет данные для последующих запросов.

Шаблоны

Шаблоны контроллеров

Один из способов использования шаблонов в Laravel - в виде шаблонов контроллеров. Если в классе контроллера определить свойство `layout`, то указанный шаблон будет создан автоматически и будет использоваться при генерации ответа клиенту.

Определение шаблона контроллера:

```
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. В отличии от шаблонов контроллеров 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 переменных

```
Привет, {{ $name }}.

Текущее время эпохи UNIX: {{ time() }}.
```

Конечно, весь пользовательский ввод должен быть экранирован или очищен. Для экранирования используйте тройные скобки:

```
Привет, {{{ $name }}}.
```

> **Внимание:** будьте очень осторожны и экранируйте переменные, которые

содержат ввод от пользователя. Всегда используйте тройные скобки, чтобы преобразовать HTML-сущности в переменной в текст.

Как показывает практика, вместо экранирования только пользовательских переменных безопаснее экранировать весь вывод, делая исключения только в редких случаях. Некоторые альтернативные шаблонизаторы, такие как HTMLki, делают это автоматически - прим. пер.

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
```

Языковые строки

```
@lang('language.line')

@choice('language.line', 1);
```

Комментарии

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

Юнит-тесты

Введение

Laravel построен с учётом тестирования. Фактически, поддержка PHPUnit доступна по умолчанию, а файл `phpunit.xml` уже настроен для вашего приложения. В дополнение к PHPUnit Laravel также использует компоненты Symfony HttpKernel, DomCrawler и BrowserKit для проверки и манипулирования шаблонами для симуляции работы браузера.

Папка `app/tests` уже содержит файл теста для примера. После установки нового приложения Laravel просто выполните команду `phpunit` для запуска процесса тестирования.

Определение тестов для выполнения

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 через командную строку командой `phpunit`.

> **Внимание:** 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 маршрутам

Вы можете легко вызвать любой ваш маршрут методом `call`:

Вызов routing маршрута из теста:

```
$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());
```

Вы также можете вызвать из теста любой контроллер.

Вызов контроллера из теста:

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

Метод `getContent` вернёт содержимое-строку ответа маршрута 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');
```

Внимание: фильтры маршрутов отключены в тестовой среде. Для их включения

добавьте в тест вызов `Route::enableFilters()`.

DOM Crawler

Вы можете вызвать ваш маршрут 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 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.

Тестирование фасада `Event`:

```
public function testGetIndex()
{
    Event::shouldReceive('fire')->once()->with(array('name' => 'Дейл'));

    $this->call('GET', '/');
}
```

> **Внимание:** 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);
}
```

Вспомогательные методы

Класс `TestCase` содержит несколько вспомогательных методов для упрощения тестирования вашего приложения.

Вы можете установить текущего авторизованного пользователя с помощью метода `be`:

Установка текущего авторизованного пользователя:

```
$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.

Проверка ввода

Основы использования

Laravel поставляется с простой, удобной системой проверки ввода и получения сообщений об ошибках - классом `Validation`.

Простейший пример проверки ввода:

```
$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.

Несколько правил могут быть разделены либо прямой чертой (|), либо быть отдельными элементами массива.

Использование массивов для указания правил:

```
$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 Bootstrap.

Получение всех сообщений в заданном формате:

```
foreach ($messages->all('<li>:message</li>') as $message)
{
    //
}
```

Ошибки и шаблоны

Как только вы провели проверку вам понадобиться простой способ, чтобы передать ошибки обратно в шаблон. Laravel позволяет удобно сделать это. Представьте, что у нас есть такие правила:

```
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 переменных сессии, таким образом делая их доступными для следующего запроса.

Однако мы не всегда должны явно передавать сообщения об ошибках в наших GET-маршрутах. Laravel проверяет данные сессии на наличие сообщений и автоматически привязывает их к 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
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Before (Date)
- Between
- Confirmed
- Date
- Date Format
- Different
- E-Mail
- Exists (Database)
- Image (File)

- In
- Integer
- IP Address
- Max
- MIME Types
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required Without
- Same
- Size
- Unique (Database)
- URL

accepted

Поле должно быть в значении *yes*, *on* или *1*. Это полезно для проверки принятия правил и лицензий.

active_url

Поле должно быть корректным URL, доступным через функцию `checkdnsrr`.

after:*date*

Поле должно быть датой, более поздней, чем *date*. Строки приводятся к датам функцией `strtotime`.

alpha

Поле можно содержать только 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.

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 менее *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 присутствуют все перечисленные поля (*foo*, *bar* и т.д.).

required_without:*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.

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);
```

> **Внимание:** 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',
);
```

Иногда вам может потребоваться указать своё сообщение для отдельного поля.

Указание собственного сообщения для отдельного поля:

```
$messages = array(
    'email.required' => 'Нам нужно знать ваш e-mail адрес!',
);
```

Может быть также полезно определять эти сообщения в языковом файле вместо того, чтобы передавать их в `Validator` напрямую. Для этого добавьте сообщения в массив `custom` языкового файла `app/lang/xx/validation.php`.

Указание собственных сообщений в языковом файле:

```
'custom' => array(
    'email' => array(
        'required' => 'Нам нужно знать ваш e-mail адрес!',
    ),
),
```

Собственные правила проверки

Laravel изначально содержит множество полезных правил, однако вам может понадобиться создать собственные. Одним из способов зарегистрировать произвольное правило - через метод `Validator::extend`.

Регистрация собственного правила:

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

> **Внимание:** 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 замыкания в метод `extend` также 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 языковой файл.

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`.

Расширение класса `Validator`:

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'значение';
    }

}
```

Затем вам нужно зарегистрировать собственное расширение.

Регистрация нового класса `Validator`:

```
Validator::resolver(function($translator, $data, $rules, $messages)
{
    return new CustomValidator($translator, $data, $rules, $messages);
});
```

Иногда при создании своего класса вам может понадобиться определить собственные строки-переменные 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);
}
```

Расширение фреймворка

Введение

Laravel предоставляет вам множество точек для настройки поведения различных частей ядра библиотеки или даже полной замены. К примеру, хэширующие функции определены контрактом (интерфейсом) `HasherInterface`, который вы можете реализовать 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 и регистрация расширения через класс `Manager`, который реализует шаблон проектирования "Фабрика". В этом разделе мы изучим различные методы расширения фреймворка и код, который для этого необходим.

> **Подсказка:** компоненты Laravel обычно расширяются через IoC или классы `Manager` - они служат реализацией "фабрик" и ответственны за работу подсистем, основанных на драйверах, таких как кэширование и сессии.

Управляющие и фабрики

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.

> **Подсказка:** 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.

Кэш

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" мы бы сделали 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` и IoC-контейнер.

Для создания стороннего драйвера для кэша мы начнём с реализации контракта (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 драйвер - подумайте о публикации 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`. Однако держите в уме то, что Laravel не имеет жёсткой структуры папок и вы можете рганизовать свои файлы, как вам удобно.

> **Подсказка:** если у вас стоит вопрос о том, где должен располагаться определённый код, в первую очередь вспомните о поставщиках услуг. Как мы уже говорили, их использование для упорядочивания расширений библиотеки очень здорово упорядочивает их код.

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
});
```

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 понадобиться добавлять что-либо в этот метод. Вы можете всегда оставить его пустым. Фактически, это просто плохое решение 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` (отпечатка времени Unix). Для самоочищающихся систем вроде Memcached и Redis этот метод может быть пустым.

Как только интерфейс `SessionHandlerInterface` был 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!

Авторизация

Механизм авторизации 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 автоувеличивающееся числовое поле ID в СУБД 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 `UserProviderInterface`давайте 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 вы 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 его в вашем файле настроек `app/config/auth.php`.

Расширения посредством IoC

Почти каждый поставщик услуг Laravel получает свои объекты из контейнера IoC. Вы можете увидеть список поставщиков 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, `PaginationServiceProvider` использует ключ `paginator` для получения экземпляра `Illuminate\Pagination\Environment` из контейнера IoC. Вы можете легко расширить и перекрыть этот класс в вашем приложении перекрыв эту привязку. К примеру, вы можете создать класс, расширяющий `Environment`:

```
namespace Snappy\Extensions\Pagination;

class Environment extends \Illuminate\Pagination\Environment {

    //

}
```

Как только вы написали расширение можно создать нового поставщика услуг `SnappyPaginationProvider, который перекроет объект страничного вывода (paginator) в`

своём методаboot`:

```
class SnappyPaginationProvider extends PaginationServiceProvider {

    public function boot()
    {
        App::bind('paginator', function()
        {
            return new Snappy\Extensions\Pagination\Environment;
        });

        parent::boot();
    }

}
```

Заметьте, что этот класс расширяет `PaginationServiceProvider`, а не класс `ServiceProvider` по умолчанию. Как только вы расширили этого поставщика, измените `PaginationServiceProvider` в файле настроек `app/config/app.php` на имя вашего нового поставщика услуг.

Это общий подход к расширению любого класса ядра, который привязан к контейнеру. Фактически каждый класс так или иначе привязан к нему, и с его помощью может быть перекрыт. Опять же, прочитав код включённых в библиотеку поставщиков услуг вы познакомитесь с тем, где различные классы привязываются к контейнеру и какие ключи для этого используются. Это отличный способ понять глубинную работу Laravel.

Расширение запроса

Расширение класса `Request` происходит немного иначе в отличии от описанных ранее из-за того, что это основополагающая часть фреймворка и создаётся в самом начале обработки запроса.

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 нескольких первых 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 контейнеру IoC 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 иметь под рукой собственную реализацию, даже в юнит-тестах!

Основы работы с базами данных

Настройка

Laravel делает процесс соединения 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.

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');
```

Подсказка: методы `update` и `delete` возвращают число затронутых строк.

Выполнение запроса другого типа:

```
DB::statement('drop table users');
```

Вы можете добавить собственный обработчик, вызываемый при выполнении очередного запроса, с помощью метода `DB::listen`:

Реагирование на выполнение запросов:

```
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) 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.

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 для этого подключения:

```
$pdo = DB::connection()->getPdo();
```

Иногда вам может понадобиться переподключиться и вы можете сделать это так:

```
DB::reconnect('foo');
```

Журнал запросов

По умолчанию Laravel записывает все SQL-запросы, выполненные в рамках текущего запроса страницы. 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();
```

Конструктор запросов

Введение

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.

> **Внимание:** конструктор запросов Laravel использует привязку параметров к запросам средствами 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();
```

Фильтрация через 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();
```

Сложная фильтрация (WHERE)

Иногда вам нужно сделать выборку по более сложным параметрам, таким как "существует ли" или вложенная группировка условий. Конструктор запросов Laravel справится и с такими запросами.

Группировка условий:

```
DB::table('users')
            ->where('name', '=', 'Джон')
            ->orWhere(function($query)
            {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Админ');
            })
            ->get();
```

Команда выше выполнит такой SQL:

```
select * from users where name = 'Джон' or (votes > 100 and title <> 'Админ')
```

Проверка на существование:

```
DB::table('users')
            ->whereExists(function($query)
            {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

Эта команда выше выполнит такой запрос:

```
select * from users
```

```
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

Агрегатные функции

Конструктор запросов содержит множество агрегатных методов, таких как `count`, `max`, `min`, `avg` и `sum`.

Использование агрегатных 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 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)
);
```

Если таблица имеет автоматическое числовое поле (autoincrementing), то можно использовать метод `insertGetId` для вставки записи и получения её порядкового номера.

Вставка записи и получение её нового ID:

```
$id = DB::table('users')->insertGetId(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

> **Внимание:** при использовании PostgreSQL автоматическое 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();
```

Усечение таблицы:

```
DB::table('users')->truncate();
```

> Усечение таблицы аналогично удалению всех её записей, а также сбросом счётчика autoincrement -полей. - прим. пер.

Слияние (UNION)

Конструктор запросов позволяет создавать слияния двух запросов вместе.

Выполнение слияния:

```
$first = DB::table('users')->whereNull('first_name');

$users = DB::table('users')->whereNull('last_name')->union($first)->get();
```

Также существует метод `unionAll` с аналогичными параметрами.

Кэширование запросов

Вы можете легко закэшировать запрос методом `remember`:

Кэширование запросов:

```
$users = DB::table('users')->remember(10)->get();
```

В этом примере результаты выборки будут сохранены в кэше на 10 минут. В течении этого времени данный запрос не будет отправляться к СУБД - вместо этого результат будет

получен из системы кэширования, указанной по умолчанию в вашем файле настроек.

Eloquent ORM

Введение

Система объектно-реляционного отображения ORM Eloquent - красивая и простая реализация шаблона 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`.

Простейшее использование

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';

}
```

> **Внимание:** 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);
```

> **Внимание:** 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);
}
```

Конечно, вам также доступны агрегатные функции.

Агрегатные функции в Eloquent:

```
$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();
```

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 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.

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 её свойство `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 формата 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 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 получить к нему доступ через динамические свойства Eloquent:

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

Сгенерированный SQL имеет такой вид:

```
select * from users where id = 1

select * from phones where user_id = 1
```

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 упорядоченных по алфавиту имён связанных моделей и она 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, 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 возвращён 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 обеих

моделях:

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

foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

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

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 верните 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.

Setting A Model Boot Method

```
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 часто 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

В Laravel, 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($table)
{
    $table->increments('id');
});
```

Первый параметр метода `create` - имя таблицы, а второй - замыкание, которое получает объект `Closure`, использующийся для заполнения новой 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($table)
{
    $table->increments('id');
});
```

Для удаления таблицы вы можете использовать метод `Schema::drop`:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Добавление полей

Для обновления существующей таблицы мы будем использовать метод `Schema::table`:

```
Schema::table('users', function($table)
{
    $table->string('email');
});
```

Конструктор таблиц поддерживает различные типы полей, которые вы можете использовать при создании таблиц.

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

Команда	Описание
`->default($value)`	Указывает значение по умолчанию для поля
`->unsigned()`	Обозначает беззнаковое число UNSIGNED

Если вы используете MySQL, то метод `after` позволит вам вставить поле после определённого существующего поля.

Вставка поля после существующего в MySQL:

```
$table->string('name')->after('email');
```

Переименование полей

Для переименования поля можно использовать метод `renameColumn` объекта конструктора.

Переименование поля

```
Schema::table('users', function($table)
{
    $table->renameColumn('from', 'to');
});
```

> **Внимание:** переименование полей типа `enum` не поддерживается.

Удаление полей

Удаление одного поля из таблицы:

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes');
});
```

Dropping Multiple Columns From A Database Table

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes', 'avatar', 'location');
});
```

Проверка на существование

Вы можете легко проверить существование таблицы или поля с помощью методов `hasTable` и `hasColumn`.

Проверка существования таблицы:

```
if (Schema::hasTable('users'))
{
    //
}
```

Проверка существования поля:

```
if (Schema::hasColumn('users', 'email'))
{
    //
}
```

Добавление индексов

Конструктор таблиц поддерживает несколько типов индексов. Есть два способа добавлять индексы: можно определять их на самих полях, либо добавлять отдельно.

Определение индекса на поле:

```
$table->string('email')->unique();
```

Вы можете добавлять их отдельно. 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.

Добавление внешнего ключа:

```
$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 автоматическое 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"

Системы хранения

Для задания конкретной системы хранения таблицы установите свойство `engine` объекта конструктора:

```
Schema::create('users', function($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 - что-то вроде системы контроля версий для вашей базы данных. Они позволяют команде изменять её структуру, в то же время оставаясь в курсе изменений других участников. Миграции обычно идут рука об руку с построителем структур для более простого обращения с архитектурой вашего приложения.

Создание миграций

Для создания новой миграции вы можете использовать команду `migrate:make` командного интерфейса Artisan.

Создание миграции:

```
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.

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 - прим. пер.).

```
php artisan migrate:make create_users_table --table=users --create
```

Применение миграций

Накатывание всех неприменённых миграций:

```
php artisan migrate
```

Накатывание новых миграций 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 update`.

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, `UserTableSeeder`и т.д. По умолчанию для вас уже определён класс `DatabaseSeeder`. Из этого класса вы можете вызывать

метод `call` для подключения других классов с данными, что позволит вам контролировать порядок их выполнения.

Примерные классы для загрузки начальных данных:

```
class DatabaseSeeder extends Seeder {

    public function run()
    {
        $this->call('UserTableSeeder');

        $this->command->info('Таблица пользователей загружена 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 команду `db:seed` Artisan:

```
php artisan db:seed
```

Либо вы можете сделать это командой `migrate:refresh`, которая также откатит и заново применит все ваши миграции:

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

Redis

Введение

Redis - открытое продвинутое хранилище пар ключ/значение. Его часто называют сервисом структур данных, так как ключи могут содержать строки, хэши, списки, наборы, and сортированные наборы.

> **Внимание:** сли 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),

),
```

Значения по умолчанию должны подойти для разработки. Однако вы свободно можете менять этот массив в зависимости от своего окружения. Просто 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 to the instance.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));
```

Когда вы выполняете команды на подключении по умолчанию просто вызывайте соответствующие статические методы класса `Redis`:

```
Redis::set('name', 'Тейлор');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

Внимание: Laravel поставляется с драйверами Redis для кэширования и сессий.

Конвейер

Конвейер 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 интерфейса командной строки, с которым поставляется Laravel. Он содержит набор полезных команд, помогающие вам при разработке приложения. Он основан на мощном компоненте Symfony Console.

Использование

Для просмотра списка доступных команд используйте команду `list`.

Список всех доступных команд:

```
php artisan list
```

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`.

Помощь по выбранной команде:

```
php artisan help migrate
```

Вы можете указать среду, которая будет использоваться при выполнении команды, с помощью ключа `--env`.

Указание имени среды для выполнения команды:

```
php artisan migrate --env=local
```

Вы также можете узнать версию текущей установки Laravel с помощью ключа `--version`:

Версия текущей установки Laravel:

```
php artisan --version
```

Команды Artisan

Введение

В дополнение к стандартным командам Artisan вы можете также добавлять свои собственные команды для работы с приложением. Вы можете поместить их в папку `app/commands`, либо в любое другое место, в котором их смжет найти автозагрузчик в соответствии с вашим файлов `composer.json`.

Создание команды

Создание класса

Для создания новой команды можно использовать команду `command:make`, которая создаст заглушку, с которой вы можете начать работать.

Генерация нового класса команды:

```
php artisan command:make FooCommand
```

По умолчанию сгенерированные команды помещаются в папку `app/commands`, однако вы можете указать произвольное расположение или пространство имён:

```
php artisan command:make FooCommand --path=app/classes --namespace=Classes
```

Написание команды

Когда вы сгенерировали класс команды, вам нужно заполнить его свойства `name` и `description` , которые используются при отображении вашей команды в списке команд `list`.

Метод `fire` будет вызван при вызове вашей команды. В него вы можете поместить любую нужную логику.

Параметры и ключи

Методы `getArguments` и `getOptions` служат для определения параметров и ключей, которые ваша команда принимает на вход. Оба эти метода могут возвращать массив команд, которые описываются как массив ключей.

При определении `arguments` массив имеет такую форму:

```
array($name, $mode, $description, $defaultValue)
```

Параметр `mode` может быть объектом `InputArgument::REQUIRED` или `InputArgument::OPTIONAL`.

При определении `options`, массив выглядит так:

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

`mode` для ключей может быть любым из этих объектов: `InputOption::VALUE_REQUIRED`, `InputOption::VALUE_OPTIONAL`, `InputOption::VALUE_IS_ARRAY`, `InputOption::VALUE_NONE`.

Режим `VALUE_IS_ARRAY` указывает, что ключ может быть передан несколько раз:

```
php artisan foo --option=bar --option=baz
```

Режим `VALUE_NONE` указывает, что ключ является простым переключателем:

```
php artisan foo --option
```

Чтение ввода

Во время выполнения команды вам, конечно, потребуется доступ к параметрам и ключам, которые были переданы ей на вход. Для этого вы можете использовать методы `argument` и `option`.

Чтение параметра команды:

```
$value = $this->argument('name');
```

Чтение всех параметров:

```
$arguments = $this->argument();
```

Чтение ключа команды:

```
$value = $this->option('name');
```

Чтение всех ключей:

```
$options = $this->option();
```

Вывод

Для вывода сообщений вы можете использовать методы `info`, `comment`, `question` и `error`. Каждый из них будет использовать подходящие цвета ANSI при отображении текста.

Вывод сообщения в консоль:

```
$this->info('Показать это на экране');
```

Вывод сообщения об ошибке:

```
$this->error('Что-то пошло не так!');
```

Запрос ввода

Вы можете также использовать методы `ask` и `confirm` для получение ввода от пользователя.

Запрос ввода от пользователя:

```
$name = $this->ask('Как вас зовут?');
```

Запрос скрытого ввода:

```
$password = $this->secret('Какой пароль?');
```

Запрос подтверждения:

```
if ($this->confirm('Вы хотите продолжить (да/нет)?'))
{
    //
}
```

Вы можете также передать значение по умолчанию в метод `confirm`, которое должно быть либо `true`, либо `false`:

```
$this->confirm($question, true);
```

Регистрация команд

Как только ваша команда написана вам нужно зарегистрировать её в Artisan, чтобы она стала доступна для использования. Это обычно делается в файле `app/start/artisan.php`. Внутри него вы можете использовать метод `Artisan::add` для регистрации команд.

Регистрация команды Artisan:

```
Artisan::add(new CustomCommand);
```

Если ваша команда зарегистрирована внутри контейнера IoC, то вы можете использовать метод `Artisan::resolve`, чтобы сделать её доступной для Artisan.

Регистрация команды, находящейся в IoC:

```
Artisan::resolve('binding.name');
```

Вызов других команд

Иногда вам потребуется вызвать другую команду изнутри вашей. Вы можете сделать это методом `call`.

Calling Another Command

```
$this->call('command.name', array('argument' => 'foo', '--option' => 'bar'));
```