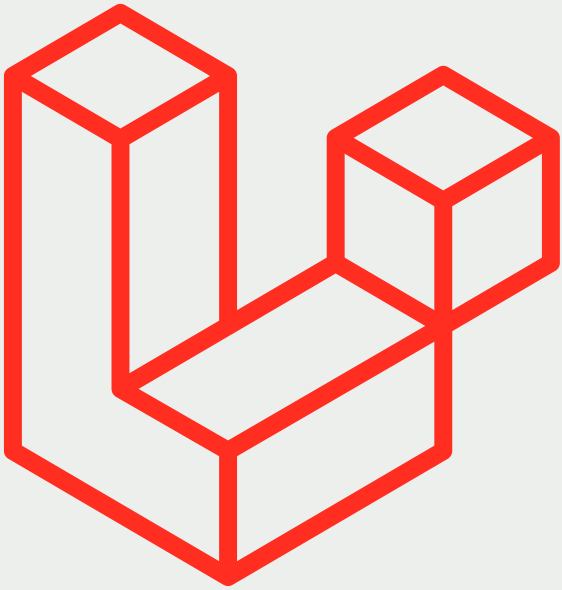
CutCode

ДАНИЛ ЩУЦКИЙ

LARAVEL NINJA GUIDE



v. 1.2

ОГЛАВЛЕНИЕ

БРИФ

Небольшое вступление о подаче материала.
Сразу обозначу, что эта книга - стартовый гайд для тех разработчиков, которые или не знакомы с Laravel, или слабо знакомы, но при этом имеют фундаментальные знания в php и базах данных. Перед вами гайд, который поможет быстро влиться в эту среду.

Такой стиль выбран не случайно, поскольку большинство читателей - это новички в Laravel. Я тоже был новичком и прекрасно помню, как на первых этапах нами правят мечты и фантазии о мире разработке, которые быстро разбиваются о недружественные термины и скучную документацию Laravel.

Поэтому при написании книги ставил целью сделать процесс обучения комфортным, не перегружая материал сложными фразами и огромным количеством материала. Это только отпугнет новичков. Я буду стараться использовать стиль аналогий и максимально просто объяснять материал, чтобы ваше погружение в мир Laravel было комфортным. Тогда начинающие разработчики смогут быстро понять концепции этого прекрасного фреймворка. А получив базовые знания, далее с опытом и практикой вы будете все сильнее углубляться в него, изучать более сложные понятия и концепции. Но всему свое время.

Итак, основная цель гайда (звучит крайне страшно) - понизить популяцию «страус-программистов».

Кто же такой «страус-программист»? Тот, кто раньше времени задрал голову, выполнил пару проектов (зачастую просто повторив код из видеоуроков) и считает, что уже стал профессионалом. Мы же будем учиться другим способом: в правильном порядке изучать все возможности Laravel и постепенно повышать свой

уровень знаний, чтобы в итоге уверенно владеть этим инструментом и стать настоящим Laravel-ninja.

Я не люблю термин «программист», он не имеет души. А вот «разработчик» уже звучит по другому. Разработчик - творец и это действительно так, ведь перед нами огромный мир, который мы создаем именно так, как видим (при этом крайне желательно использовать общепринятые принципы и паттерны).

В книге будет описана работа с фреймворком Laravel самой свежей версии - 10, но большая часть информации будет актуальна и для более старых версий.
В свою очередь настоятельно рекомендую вам своевременно обновлять версию фреймворка!

Для полного погружения в обучение предлагаю при изучении книги использовать Laravel Roadmap от CutCode:

https://cutcode.dev/roadmap

Roadmap отлично дополнит этот гайд и сделает обучение интерактивным. Читайте книгу, продвигайтесь по дорожной карте и отмечайте изученные темы!

Вместе с Вами по страницам этой книги будет передвигаться маскот моего проекта CutCode - пес Альфа. Он будет постепенно развиваться, как и Вы, учиться и получать все более высокие звания в искусстве web-разработки.

УРОВЕНЬ

NULL

ДОКУМЕНТАЦИЯ

> *«Основная ошибка страус-программиста - полное игнорирование официальной документации Laravel. Впрочем, если подумать, то она страус-программистам и не нужна, они каждый раз изобретают велосипед заново, пишут свою систему маршрутизации, в общем активно и бессмысленно работают. Мы же пользуемся уже существующими инструментами и решениями для базовых вещей, и не тратим времени на их реализацию. При этом получаем безопасный рабочий функционал с подробными инструкциями»*

Документация во время обучения и работы - постоянный источник актуальной и корректной информации. Даже разработчики с большим опытом и полным понимаем Laravel продолжают обращаться к документации, и это нормально! Мы в этом гайде также постоянно будем обращаться к документации, чтобы превратить это в полезную привычку. Давайте и начнем с этой полезной ссылки (предлагаю добавить в закладки):

 https://laravel.com/docs

УСТАНОВКА И ЗАПУСК

 https://laravel.com/docs/installation

> *«Страус-программист копирует файлы от проекта к проекту, скачивает и потом распаковывает архивы - ужас. Laravel-разработчик ленив и не тратит время на бесполезную суету. Он использует минимум усилий и менеджер пакетов composer и устанавливает все одной командой. Есть еще более изящные способы установки, которые вы увидите в документации, но чтобы не пугать вас дополнительными темами, не имеющими отношения к Laravel, мы начнем с composer»*

Ах да, друзья, уточняю, что php, composer и система управления базой данных (например, mysql или postgresql), у нас уже должны быть установлены к моменту развертывания Laravel. Если нет - то не отчаивайтесь и отправляйтесь в гугл, чтобы узнать, как установить php и sql для своей операционной системы. И я бы рекомендовал для начала изучить эти языки, а потом уже пользоваться прекрасным инструментом - Laravel. Без знания нативного PHP вы наверняка разочаруетесь в Laravel, так как просто не поймете его функционал.

Итак, мы установили все необходимое и выполняем простую команду в терминале IDE (командная строка) в директории, где будет располагаться новый проект. Я к примеру у себя назвал эту директорию `Projects`:

```
composer create-project laravel/laravel example-app
```

Тем самым мы загружаем новый чистый проект с Laravel с названием `example-app`. Название можете указать любое, более подходящее для вашего проекта - оно будет служить названием директории с проектом. В моем случае это `Projects/example-app`.

А далее переходим в эту директорию и запускаем виртуальный сервер, для нашего гайда этого будет достаточно:

```
cd example-app
php artisan serve
```

В консоли появится ссылка на наше приложение, после перехода по которой нас встретит стартовая страница нашего первого проекта на Laravel. Мы только что встали с вами на путь ninja Laravel-разработчика, уже чувствуете это прекрасное ощущение легкости и новых возможностей? Готовы примерить на себя белый пояс Laravel ninja? Тогда начнём. И первым делом разберемся, что это за artisan такой.

Посмотреть как устанавливать Laravel можно в этом видео

 https://www.youtube.com/watch?v=cD247LTT9Dw

ARTISAN

> *«Страус-программист медлительный, он раз за разом вручную копирует классы. Тем временем Laravel-разработчик делает все одной командой, и в этом ему помогает artisan»*

Artisan это файл, находящийся в корне нашего Laravel проекта. Он запускает наше Laravel приложение в консольном режиме и помогает взаимодействовать с ним не через http запросы, а через терминал, что очень удобно.

С помощью artisan можно запускать множество команд, упрощающих нам работу. Например, вручную создавать контроллер или модель, нам достаточно просто выполнить команду подобно этой:

```
php artisan make:controller MyFirstController
```

Ура, мы создали новый контроллер с именем `MyFirstController`.

Друзья, если я уже напугал вас терминами контроллер или модель, не отчаивайтесь, мы все это разберем. Сейчас я просто хочу донести, что благодаря artisan и консольным командам Laravel, мы можем делать все быстрее. А это очень важно.

Чтобы ознакомиться со всеми доступными командами - выполните в терминале:

```
php artisan list
```

УРОВЕНЬ 1.

БЕЛЫЙ ПОЯС

ГЛАВА 1. ПРОГУЛКИ ПО ПУТИ ЗАПРОСА

https://laravel.com/docs/lifecycle

> *«У запроса есть путь, а также цель - сделать вас Laravel-разработчиком»*

Друзья, мы начинаем путешествие с классического веб-приложения Laravel и его основа - http запросы. Как только мы вбиваем в браузер адрес нашего приложения, мы запускаем путь запроса внутрь нашего проекта с Laravel и сейчас подробно об этом поговорим. Понимание как устроены запросы - основа веб-проекта, поэтому мы начинаем именно с этой темы. И помните, что не только вы, но и каждый клиент вашего проекта будет проходить точно такой же путь.

Итак, мы отправляемся в путь вместе с нашим запросом. Откройте браузер, обновите страницу с нашим новым Laravel-проектом, и как только вы это сделаете, http запрос отправляется в путь по фреймворку! И точка входа нашего запроса - `public/index.php`, отсюда начинается наше веб приложение.

Шаг за шагом мы вместе с запросом пройдём каждую строку в файле `public/index.php`. Откройте его, чтобы двигаться вместе со мной и нашим запросом.

Первое на что стоит уделить внимание:

```
require __DIR__.'/../vendor/autoload.php';
```

Все необходимые пакеты и сам фреймворк уже установлены, но Laravel еще не запущен:

```
$app = require_once __DIR__.'/../bootstrap/app.php';
```

Этой командой мы запускаем (правильно говорить инициализируем) Laravel приложение. Одновременно мы активировали service

container, это важная, но сложная деталь в Laravel, и мы обязательно ее изучим.

А сейчас кратко посмотрим на возможности service container в следующей строке `index.php`:

```
$app = require_once __DIR__.'/../bootstrap/app.php';
```

Прямо сейчас мы начинаем запуск Laravel-приложения. И мы можем выбрать, когда нам запускать веб приложение, а когда консольное. Давайте взглянем на код:

```
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
);
```

Нас встречает service container Laravel. Давайте попробуем понять, что тут происходит. Обратите внимание на php интерфейсы `Illuminate\Contracts\Http\Kernel` и `Illuminate\Contracts\Console\Kernel`.

Мы указали Laravel, что php interface `Illuminate\Contracts\Http\Kernel` у нас относится к веб-приложению и ссылается на класс `App\Http\Kernel`. Ну а `Illuminate\Contracts\Console\Kernel` ссылается на консольное приложение, с которым мы взаимодействуем через artisan.

А теперь развязка. Смотрим на следующую строку index.php и мы видим, что создаем Laravel приложение на основе интерфейса `Illuminate\Contracts\Http\Kernel`:

```
$kernel = $app->make(Kernel::class);
```

В `bootstrap/app.php`, который мы недавно смотрели, мы указали, что в рамках этого интерфейса мы будем работать с http запросами (веб-приложение). Это одна из особенностей service container - мы можем сослаться на конкретный класс через интерфейс, если потребуется менять реализацию и пользоваться возможностями DI (инъекции зависимостей) без привязки к конкретным классам и инициализируя их на лету.

Ради интереса откройте также `artisan.php` и вы увидите:

```
$kernel = $app->make(Illuminate\Contracts\Console\
Kernel::class);
```

Это тоже инициализация приложения, но уже консольного.

Возвращаемся назад в `public/index.php` и самое интересное происходит далее:

```
$response = $kernel->handle(
    $request = Request::capture()
)->send();
```

До этой команды наш запрос был запросом в «сыром» виде - это HTTP метод, заголовок и тело запроса. А вот этой строчкой кода:

```
$request = Request::capture();
```

мы превратили запрос в объект и с этим объектом нам предстоит часто встречаться при разработке на Laravel.

Ну а мы двигаемся дальше в метод `handle`. Сам метод выглядит вот так:

```
public function handle($request)
{
    try {
        $request->enableHttpMethodParameterOverride();

        $response = $this->sendRequestThroughRouter(
$request);
    } catch (Throwable $e) {
        $this->reportException($e);

        $response = $this->renderException($request, $e);
    }

    $this->app['events']->dispatch(
        new RequestHandled($request, $response)
    );

    return $response;
}
```

В этом блоке кода мы обработали наш запрос и в результате отдали ответ с определенной страницей нашего сайта в браузере, но прежде мы прошли через важный шаг:

```
$response = $this->sendRequestThroughRouter($request);
```

Давайте взглянем подробнее на этот метод:

```
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);

    Facade::clearResolvedInstance('request');

    $this->bootstrap();

    return (new Pipeline($this->app))
                ->send($request)
                ->through($this->app->shouldSkipMiddleware()
? [] : $this->middleware)
                ->then($this->dispatchToRouter());
```

Здесь нас встречают два важных шага:

1) `$this->bootstrap();`

   Тут мы запустили нужные для фреймворка службы, фасады, провайдеры, конфиги и настройки окружения, о которых мы еще поговорим в этом гайде.

2) `return (new Pipeline($this->app))`

Ух, друзья мы дошли до Pipeline и я их обожаю. Они часто встречаются внутри фреймворка Laravel и со временем и вы их полюбите. Но для начала давайте немного в них разберемся, иначе понять тему пути запроса будет сложно.

Pipeline переводится как трубопровод и это хорошо описывает суть этой концепции. Мы передаем в pipeline определенный объект и указываем набор труб (обработчиков), через которые он пройдет. Каждая труба может модифицировать или не пропустить дальше этот объект. И на выходе мы получим ту же самую (или модифицированную) версию объекта, который изначально был отправлен в

pipeline:

```
return (new Pipeline($this->app))
                ->send($request)
                ->through($this->app->shouldSkipMiddleware()
? [] : $this->middleware)
                ->then($this->dispatchToRouter());
}
```

В данном примере в роли объекта, который мы отправляем по трубопроводу, выступает наш запрос (`send($request)`).

Далее с помощью метода `through` мы указываем набор труб, через которые он пройдет, и в нашем случае это мидлвары (Middleware). Запрос пройдет по всем переданным мидлварам, от одного к другому, и каждый из них проверит наш запрос в рамках своих требований. При этом некоторые мидлвары и вовсе могут не пустить запрос дальше и ответить редиректом или 403 ошибкой. В следующей главе мы с вами рассмотрим их, а также последнюю важную команду `->then($this->dispatchToRouter());`.

Мы рассмотрели путь, который проходит каждый запрос. Это большая и важная тема. Не случайно именно она находится в самом начале этой книги. Для более глубокого понимания настоятельно рекомендую посмотреть видео на моём канале:

 https://www.youtube.com/watch?v=LZqEMQgpYWg

ГЛАВА 2. MIDDLEWARE

https://laravel.com/docs/middleware

> *«Страус-программист дублирует код, каждый раз добавляя проверки для того, чтобы понять, как поступить с запросом. Laravel-разработчик создает мидлвары и управляет запросами с их помощью»*

В предыдущей главе мы дошли до мидлваров и знаем, что наш запрос пройдет через множество таких мидлваров, каждый из которых осуществит проверку и возможную фильтрацию запроса. Что же такое мидлвар? Это как раз середина пути нашего запроса. Давайте взглянем на один из мидлваров чтобы понять их суть. Хотя нет! Лучше мы создадим собственный мидлвар и посмотрим, как это делается. Для этого воспользуемся artisan и выполним команду:

```
php artisan make:middleware FirstMiddleware
```

Мидлвар создан. Внутри нас ждет простой класс с главным ключевым методом `handle`:

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

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

В аргументах у нас текущий запрос и аргумент со следующим мидлваром. Благодаря этому они выполняются по порядку: в методе `handle` вызывается следующий по порядку мидлвар с помощью кода:

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

С помощью такой реализации метода `handle` мы и передаем за-

прос от одного middleware к другому через pipeline.

Теперь добавим логику внутрь созданного нами мидлвара. Например, если у запроса ip адрес пользователя не соответствует «190.90.90.90», тогда запрос мы дальше не пускаем и отдаем ответ с 404 ошибкой, выглядеть это будет вот так:

```
public function handle($request, Closure $next)
{
    if ($request->ip() !== '190.90.90.90') {
        abort(404);
    }

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

Как видите, при помощи middleware мы можем проверить запрос: на правильность заголовков, наличия определенных параметров, например ip адреса клиента как в примере выше. Один middleware может проверять авторизован ли пользователь, и если нет, то не пустит запрос дальше. Другой middleware будет проверять роли пользователей для выдачи доступа к определенным страницам. Также middleware часто используют не по прямому предназначению (фильтрация и проверка запроса), но и для того, чтобы установить сессию, запоминая ряд параметров. Таким образом мы можем только один раз передать язык пользователя в url, а в дальнейшем этот параметр будет браться из сессии.

Зная что из себя представляет сам мидлвар, давайте посмотрим, где располагается их список. Там уже есть набор мидлваров «из коробки» и мы можем добавлять туда новые (как пример только что созданный).

Сам список всех мидлваров, с которыми придется столкнуться нашему запросу, располагается здесь - `app/Http/Kernel.php` :

```
protected $middleware = [
    // \App\Http\Middleware\TrustHosts::class,
    \App\Http\Middleware\TrustProxies::class,
    \Illuminate\Http\Middleware\HandleCors::class,
    \App\Http\Middleware\PreventRequestsDuringMaintenance::
class,
    \Illuminate\Foundation\Http\Middleware\
ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\
ConvertEmptyStringsToNull::class,
];
```

В самом начале класса мы видим список мидлваров, через который пройдет каждый запрос. Последовательно, сверху вниз, ни один запрос не минует этот список:

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\
AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\
ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\
SubstituteBindings::class,
    ],
    'api' => [
        // \Laravel\Sanctum\Http\Middleware\
EnsureFrontendRequestsAreStateful::class,
        'throttle:api',
        \Illuminate\Routing\Middleware\SubstituteBindings::
class,
    ],
];
```

Этот список будет зависеть от группы маршрутизации, об этом мы поговорим дальше.

Да, мы вернулись к нашему Pipeline, прошли все мидлвары и столкнулись с этой строчкой кода:

```
    →then($this→dispatchToRouter());
```

И теперь готовы перейти к важнейшей теме в Laravel - маршрутизации.

ГЛАВА 3. ROUTE

> *«Страус-программист каждый раз пишет свою роут систему, изобретая велосипед, а Laravel-разработчик использует уже готовую супер крутую роут систему под капотом»*

Давайте взглянем на метод `dispatchToRouter`.

Мы прошли еще пару методов внутри и дошли до `dispatchToRoute`:

```
public function dispatchToRoute(Request $request)
{
    return $this->runRoute($request, $this
        ->findRoute($request));
}
```

Здесь, в методе `findRoute`, мы начинаем работать с роут-системой Laravel. Роут-систему можно сравнить с сортировщиком, который изучает поступающие запросы, и по заданным критериям (правилам) распределяет их по разным направлениям. Давайте взглянем на то, где располагаются эти правила и как они выглядят.

Сейчас мы переместимся с вами в класс `app/Providers/RouteServiceProvider.php`, в котором мы объявим, где именно располагаются наши правила, чтобы разделить их по группам ответственности:

```
public function boot()
{
    $this->routes(function () {
        Route::middleware('web')
            ->group(base_path('routes/web.php'));
    });
}
```

Пока рассмотрим только группу web, чтобы не усложнять понимание.

Здесь в методе `routes` мы указали, что у нас всего одна группа правил маршрутизации, которая располагается в файле `routes/web.php`. Указав `web` в `Route::middleware`, мы как раз указали ту группу мидлваров в файле `app/Http/Kernel.php`, которую рассматривали выше:

```
'web' => [
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\
AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\
ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::
class,
],
```

Здесь используются мидлвары, которые запускают сессии для нашего веб-приложения (`StartSession`), проверяют корректность csrf, о которой мы еще поговорим (`VerifyCsrfToken`), и подключают магию для маршрутизации в `SubstituteBindings` (к ней мы тоже вернемся). А сейчас давайте посмотрим что у нас в `routes/web.php` :

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

Пока что здесь всего одно правило, которое гласит - если запрос метода GET и его урл `/` (главная страница), тогда отдай html с главной страницей.

Друзья, что такое view и на что еще способна маршрутизация, мы поговорим в процессе. Пока что мы получили представление о том, как гулял наш запрос по Laravel приложению. Если коротко, то сна-

чала наш запрос прошел таможню с мидлварами и получил билет на нужный поезд в системе маршрутизации, а потом отобразил нам главную страницу, где большими буквами нас встречает заголовок Laravel, вдохновляющий нас на продолжение приключений в этом прекрасном мире.

Подходящее для закрепления видео по теме на моём канале:

https://www.youtube.com/watch?v=i_pkBJSVFzA

УРОВЕНЬ 2.

ЖЁЛТЫЙ ПОЯС

ГЛАВА 4. СТРУКТУРА

> *«Страус-программист думает только о себе, структура его проектов понятна только ему, а в названиях часто встречаются ошибки и транслит. У Laravel-разработчика полный порядок по-умолчанию, еще до того как он начал что-то делать»*

После создания нового Laravel-проекта нас встречает уже готовая структура, давайте рассмотрим каждую директорию:

app
Сердце бизнес логики нашего приложения, здесь Controllers, Models, Action и Service классы (которых пока нет, но вам предстоит с ними познакомиться) и многое другое, о чем мы еще поговорим.

bootstrap
Помните? Гуляя по пути запроса, мы уже сюда заходили и здесь наше Laravel-приложение стартовало, включался свет для него. Кроме того, здесь будет хранится кэш файлов с конфигурациями.

config
Ничего лишнего, только настройки нашего приложения.

database
Структура базы и данные базы. Миграции, фабрики и сиды, о которых мы еще с вами поговорим.

public
Здесь находится стартовая точка нашего веб приложения - `index.php`, с которого мы начинали прогулку с запросом. А также стили, скрипты и прочие ассеты (скомпилированные стили и скрипты), необходимые нашему проекту.

lang
Локализация. Здесь мы найдем файлы с переводами приложения под те языки, которые мы будем поддерживать.

resources
Визуальное представление нашего приложения располагается здесь. Позже мы с вами еще вернемся в эту папку.

routes
Здесь мы уже были. Тут в файле `web.php` хранятся правила маршрутизации для нашего веб-приложения. А для api - в `api.php`. Или `console.php` для консольного приложения. Здесь может быть сколько угодно файлов с роутами под ваши задачи, но суть одна - помочь запросу понять, куда ему отправиться дальше.

storage
Хранилище, где Laravel хранит сессии, кэш, логи и все то, что вы будете сохранять из реквестов (как примеры пользовательские файлы и изображения).

tests
Здесь мы будем хранить тесты логики нашего приложения. Тесты нужны для того, чтобы убедиться в том, что все работает так, как нужно.

vendor
Зависимости, пакеты, библиотеки - все это и сам фреймворк. В общем всё то, что мы установили с помощью composer.

А знаете что самое прекрасное в Laravel? Что вы можете создать и свою собственную структуру проекта. Так что если вы хотите сделать большой проект и вы уже задумались о Domain Driven ... Стоп, стоп, меня понесло, не буду вас пугать, продолжаем.

Помимо директорий в рамках темы структуры, давайте также взглянем на файл `.env`. Этот файл находится в корне проекта.

Здесь у нас хранятся настройки нашего приложения. Давайте для примера рассмотрим доступ к базе данных, где мы можем указать необходимые для установления соединения реквизиты:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

Создавайте базу данных, после чего запишите ее название, а также доступы в .env, для того, чтобы мы смогли продолжить погружение в гайд и затрагивать дальнейшие темы.

ГЛАВА 5. КОНВЕНЦИЯ НАИМЕНОВАНИЙ

 https://github.com/lee-to/laravel-naming-conventions

Для упрощения взаимодействия в Laravel придуманы правила наименований. Правила нужны везде, а если правила не только знать, но и соблюдать, то это даёт значительный прирост в скорости разработки.

Laravel - наш помощник, но он будет помогать лишь в том случае, если работать по его правилам! Применяя конвенцию наименований, Laravel сможет автоматически находить нужные зависимые сущности!

По большей части конвенция наименований охватывает слой взаимодействия с базой данных.

Применение конвенции наименований в Laravel = простой, чистый и лаконичный код.

Погнали!

ОБЩИЕ ПРАВИЛА

1. Всегда в названиях используйте точное значение слова на английском. Не надо использовать транслит.

   Если не знаете перевод или правильное написание слова - просто найдите его перевод. Заодно и словарный запас расширите.

2. Избегайте ошибок в написании.

3. Старайтесь подбирать название так, чтобы сразу было понятно, что именно содержится в именовании.

4. Не сокращайте названия без острой необходимости.

```
// плохо
$acc
$conn
$descr
prepResp()
regEvtHandler()
updCom()

// хорошо
$account
$connection
$description
prepareResponse()
registerEventHandler()
updateComment()
```

ГЛАВА 5.1. ТАБЛИЦЫ БАЗЫ ДАННЫХ

Для оформления названий таблиц принято использовать имена во множественном числе с нотацией snake_case:

> *snake_case — стиль написания составных слов, при котором несколько слов разделяются символом подчеркивания. Слова как бы ползут по строке, и в результате получается дли_и_инное, как змея, название. В данном гайде snake_case всегда в нижнем регистре.*

Примеры:
`users`
`products`
`order_products`

Исключение:
Связующие таблицы для отношений `BelongsToMany`.

СВЯЗУЮЩИЕ ТАБЛИЦЫ BELONGSTOMANY

Применяем snake_case + имена двух таблиц, которые будем связывать, в единственном числе. В качестве разделителя между именами таблиц используем символ нижнего подчёркивания `_`.

Сущности между разделителем сортируются в алфавитном порядке.

Разберём на примере: есть таблицы `users` и `tasks`, приводим оба названия таблиц в единственное число и разделяем с помощью подчеркивания. Буква `t` в алфавите следует раньше `u`, поэтому результат связи двух таблиц будет - `task_user`.

Ну или пример посложнее: `order_products` и `properties`!
Результат - `order_product_property`.

Примеры:
`task_user`
`event_place`
`order_product_property`

ПОЛЯ ТАБЛИЦЫ

Все просто - snake_case. И не забывайте общие правила, о которых мы писали в одноименном разделе.

Примеры:
`created_at`
`seo_title`

PRIMARY KEY

По умолчанию Laravel ожидает `id`.

FOREIGN KEYS

snake_case в формате - имя таблицы в единственном числе, затем подчеркивание `_` и далее `Primary key ('id')`.

Формат:
`{связанная_таблица_в_единственном_числе}_{primary_key_связанной_таблицы}`

Для примера - страна пользователя, где все страны в таблице `countries`, пользователь в таблице `users` имеет foreign key для связи со страной - `country_id`.

Примеры:
`country_id`
`order_product_id`

ГЛАВА 5.2. MODELS

PascalCase в единственном числе

> *PascalCase — это стиль написания имен, при котором составные слова названия идентификаторов (в том числе и первое слово) пишутся слитно, и каждое новое слово начинается с большой буквы. Пример: MyVar, MyBestProgramm, WorkArray. Паскаль нотация используется для названий классов и публичных полей данных, а также именования процедур и функций.*

Если мы соблюдали конвенцию наименований для таблиц, то Laravel автоматически определит таблицу для модели. Как? Приведет вашу модель в snake_case в нижнем регистре, добавит подчеркивания `_` и переведёт во множественное число. Получится имя таблицы.

В итоге `User` будет ссылаться к таблице `users`, `OrderProduct` к `order_products`.

Проблемы нарушения конвенции наименований:
Если название таблицы указано не в snake_case (или название таблицы отличается от названия модели), потребуется указать в модели свойство, помогающее определить таблицу: `protected $table = 'tableName';`

тоже самое в случае с primary key - `protected $primaryKey = 'primaryKey';`

и еще масса побочных указаний для отношений)

Директория по умолчанию:
`app/Models`

Примеры:
`User`
`Product`
`OrderProduct`

КОНСТАНТЫ

При оформлении названия используем UPPER_CASE с разделителем подчеркивания.

Примеры:

```
class Document
{
    const STATUS_ACTIVE = 1;
    const STATUS_DELETE = 2;
    const STATUS_ARCHIVE = 9;
}
```

СВОЙСТВА (ПОЛЯ ТАБЛИЦЫ)

Свойства модели ссылаются на поля таблицы и имеют точно такие же наименования в snake_case.

Они становятся доступными за счет магических php методов. Обращаясь к свойствам модели, Laravel автоматически соотносит их с полями в таблице (если такие существуют).

Примеры:
`$model->created_at`
`$model->seo_title`

МЕТОДЫ (НЕ ТОЛЬКО В МОДЕЛЯХ, ВЕЗДЕ :))

При оформлении названия используем camelCase:

> *camelCase - стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом начинается с маленькой буквы, а каждое слово внутри фразы пишется с прописной буквы. И получается верблюд (camel) с его горбами.*

Примеры:
`getSomething()`

ACCESSORS/MUTATORS

Наименования поля в таблице приводится к camelCase и изменяется в методе. Отлично расписано в официальной документации.

LOCAL SCOPES

camelCase с префиксом scope. Отлично расписано в официальной документации.

ОТНОШЕНИЯ

Все просто - camelCase для модели с которой мы формируем отношение. Если отношение к одной записи, тогда именование оформляем в единственном числе, а если несколько - то во множественном числе!

Рассмотрим примеры для модели `User` с разным типом отношений к модели `Country`.

Примеры:
`belongsTo, hasOne, hasOneThrough, morphOne` = `country()`
`belongsToMany, hasMany, hasManyThrough, morphMany, morphToMany` = `countries()`

Проблемы нарушения конвенции наименований:
Может потребоваться указать `foreign_key`:
`return $this->hasOne(Phone::class, 'foreign_key');`

Может потребоваться указать `foreign_key` и `local_key`:
`return $this->hasOne(Phone::class, 'foreign_key', 'local_key');`

Может потребоваться указать и связующую таблицу и ключи:
`return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');`

ГЛАВА 5.3. МИГРАЦИИ

```
php artisan make:migration migration_name
```

Команда выше просто создаст пустой файл миграции, где методы `up` и `down` будут пустыми.

```
public function up()
{
  //
}

public function down()
{
  //
}
```

Но мы также можем влиять на содержимое миграций при их создании через artisan, если будем придерживаться правил наименования, тем самым упрощая себе жизнь.

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

Мы дали понять, что хотим создать таблицу users - `create_{Имя_таблицы}_table`. Ключевое здесь `create_{Имя_таблицы}`, а вот `table` в конце можно пропустить.

В итоге получили:

```php
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}

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

```
php artisan make:migration add_column_to_users_table
```

В данном примере мы сообщаем Laravel о нашем желании модифицировать таблицу и ключевое здесь `to_{Имя_таблицы}`.

```php
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        //
    });
}

public function down()
{
    Schema::table('users', function (Blueprint $table) {
        //
    });
}
```

ГЛАВА 5.4. ФАБРИКИ

Директория по умолчанию:
`database/factories`

Взаимодействовать с фабриками мы можем посредством метода `factory()` в моделях. Модель автоматически определит к какому классу фабрики ей ссылаться, если соблюдать конвенцию наименований.

При именовании фабрик используются те же правила, что и для моделей, только добавляем суффикс `Factory`.

Формат:
`{ИмяМодели}Factory`

Примеры:
Модель `User`, а фабрика `UserFactory`
Модель `OrderProduct`, фабрика `OrderProductFactory`

ГЛАВА 5.5. ПОЛИТИКИ (POLICY)

Директория по умолчанию:
`app/Policies`

Laravel может автоматически определить политику для модели при соблюдении конвенции наименований. В таком случае политику не требуется регистрировать «вручную».

Используем те же правила, что и для модели, только добавляем суффикс `Policy`!

Формат:
`{ИмяМодели}Policy`

Примеры:
Модель `User`, а политика `UserPolicy`
Модель `OrderProduct`, политика `OrderProductPolicy`

ГЛАВА 5.6. ПРАКТИКА НАИМЕНОВАНИЙ ОСТАЛЬНЫХ СУЩНОСТЕЙ

Остальные наименования никак не влияют на взаимодействие с Laravel, но делают код понятнее для всех кто будет с ним взаимодействовать.

ДИРЕКТОРИИ

PascalCase во множественном числе.

Примеры:
`Models`
`QueryBuilders`
`Filters`

Laravel дает свободу в расположении сущностей и наименовании директорий.

Как пример, если мы используем DDD подход , то нам никто не запрещает перенести всю бизнес логику в директорию `src`. Также перенести в `src /app` и назвать скажем `App`.

Но сами группы абстракций в хороших практиках именуются как в примере выше.

CONTROLLERS

Директория по умолчанию:
`app/Http/Controllers`

PascalCase в единственном числе с суффиксом Controller.

Примеры:
`CatalogController`
`ProductController`
`OrderController`
`BlogController`

ROUTES

snake_case либо kebab-case, пользуйтесь здравым смыслом выбирая между единственным либо множественным числом. apiResource и resource указываются во множественном числе, несмотря на то, что контроллер остается в единственном числе.

> *kebab-case стиль написания в котором слова в нижнем регистре разделяют символом дефиса. Можно представить, что слова при этом как бы насаживают на шампур — вот и получается шашлык (kebab).*

Примеры:

```
Route::get('catalog', CatalogController::class)
Route::get('categories', CategoryController::class)
Route::resource('users', UserController::class)
Route::apiResource('users', UserController::class)
```

BLADE

views директории:
snake_case во множественном числе, несмотря на то, что контроллер остается в единственном числе.

blade шаблоны:
snake_case

JSON RESPONSE

Ключи в snake_case

ГЛАВА 6. МИГРАЦИИ

> *«Страус-программист создает базу вручную через визуальные редакторы, такие как phpmyadmin. Он “бросает” в других разработчиков дампами, меняет структуру и значения записей таблиц вручную. Часто сам не помнит, когда и что он менял и менял ли вообще. Laravel-разработчик незаметно для себя хранит историю всех изменений базы данных, каждой таблицы, поля или его типа! И все вокруг радуются удобствам и аплодируют его мастерству»*

Что такое миграции?

> *Миграции — это набор функций, схожий с контролем версии БД. Миграции позволяют разработчику (команде разработчиков) определять общую схему БД и совместно использовать её. Благодаря этому инструменту все участники команды разработчиков в курсе изменений и могут развернуть структуру за секунды.*

Начнем с правил создания миграций. Они простые - на каждую таблицу и на каждое изменение таблицы мы создаем отдельную миграцию. Делается это для того, чтобы можно было легко откатить изменения и видеть всю историю взаимодействия с базой данных.

Где располагаются миграции?

`database/migrations`

Как создать миграцию?

Ничего сложного друзья! Есть несколько путей сделать это через artisan.

1) При создании модели. Когда вы создаете модель, не забудьте также добавить ей миграцию, создающую соответствующую таблицу! Модель связана с таблицей в базе, и логично сделать это сразу на этапе ее создания! Мы обсудим с вами

модели в следующей главе, а пока что просто оставим команду здесь:

```
php artisan make:model User -m
```

Параметр `-m` вместе с классом модели создаст файл миграции.

2) Напрямую просто создать миграцию командой:

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

Название миграции должно соответствовать ее содержанию. Мы создаем таблицу `users`, поэтому указали название `create_users_table` (вернитесь к предыдущей главе по конвенцией наименований в рамках миграций, чтобы понять лучше).

Пример содержимого миграции по созданию таблицы:

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateUsersTable extends Migration
{
    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('users', function (Blueprint
$table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')
                ->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Пример миграции на изменение таблицы:

```php
<?php

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

return new class extends Migration
{
    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::table('news', function (Blueprint $table) {
            $table->boolean('active')->default(true);
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::table('news', function (Blueprint $table) {
            $table->dropColumn('active');
        });
    }
};
```

Друзья, давайте немного разберемся в том, что же содержится в этих файлах с миграциями. Структурно класс миграций состоит

из двух методов: `up` и `down`. Когда мы запускаем выполнение миграций командой:

```
php artisan migrate
```

мы проходимся по всем существующим миграциям в директории `database/migrations` и в каждой запускаем метод `up`, который выполняет манипуляции с нашей базой данных. Здесь нам помогает класс `Schema`, с помощью которого мы взаимодействуем с базой данных. А в колбек функции мы передаем управление классу `Blueprint` - для взаимодействия с таблицами.

Метод `down` запускается когда мы откатываем миграции. Может случиться так, что мы что-то напутали, совершили ошибку и последнюю миграцию (или сразу несколько последних миграций) нужно откатить и вернуться к предыдущему, рабочему состоянию базы. За счет того, что мы работаем через миграции и постоянно делаем контрольные точки на каждое действие - вернуться на любой этап нам не составит проблем.

Чтобы откатить последнюю ошибочную миграцию, необходимо будет выполнить команду:

```
php artisan migrate:rollback
```

Нужно откатить 2 последних изменения? Выполняем команду:

```
php artisan migrate:rollback -step=2
```

Важно понимать, что вся история выполненных миграций сохраняется в базе данных в таблице `migrations` и тем самым Laravel знает, какие миграции мы уже выполнили и какие нужно откатить в случае с `rollback`.

Миграции в папке `migrations` выполняются по очереди, отсортированные по имени по убыванию. Поэтому для установления хронологического порядка в имена миграций по умолчанию добавляется

префикс `Timestamp` с временем и датой(`2023_02_04_240000_name`), на это стоит обращать внимание чтобы выполнять миграции в нужном порядке.

ГЛАВА 7. МОДЕЛИ

> *«Страус-программист пишет всюду sql запросы, огромное полотно кода, которое еще и дублируется из раза в раз. А у Laravel-разработчика под рукой удобный инструмент для этого»*

Модель — объект определенного класса, в свойствах которого содержаться данные полей таблицы и не только.

Модели - крайне простой и удобный инструмент в Laravel. Если говорить простыми словами, то модель - это отражение таблицы в виде объекта. Модель привязывается к таблице и мы можем обращаться к полям этой таблицы, объявлять отношения и строить запросы через модель. Это очень удобно.

Работа с моделями выполняется через магические php методы, то есть, обращаясь к свойству модели, мы обращаемся к полям таблицы (конечно если такие есть).

Пример модели, которая связана с таблицей `services`. Обратите внимание на то, насколько она простая. Но она уже способна полноценно взаимодействовать с таблицей:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Service extends Model
{

}
```

Все очень просто, если мы не нарушаем конвенцию наименований. Но если нам все-таки пришлось это сделать, то мы в модели можем указать к какой таблице она относится с помощью свойства `$table`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Service extends Model
{
    protected $table = 'custom_table';
}
```

В процессе взаимодействия с Laravel вам еще много предстоит узнать о моделях, но то, о чем точно стоит знать с самого начала, так это о свойстве `fillable`:

```php
class Service extends Model
{
    protected $fillable = [
        'title',
        'code'
    ];
}
```

При создании моделей старайтесь сразу же задавать это свойство. Оно указывает Laravel, какие именно поля модели мы можем изменять, тем самым мы заботимся о безопасности и не даем изменять те поля, которые не стоит трогать.

Рассмотрим пример. Какой-то резвый новичок поторопился и решил сохранить в базу все что пришло с реквестом:

```php
$user->create(request()->all());
```

Тем самым у злоумышленника появляется возможность заполнять любые поля вашей таблицы. И если он угадает названия полей (а они очень часто одинаковые на проектах), то сможет изменить роль пользователя на админа.

Поэтому, разрабатывая проекты, всегда страхуйтесь и думайте о безопасности.

Более подробный гайд о моделях вы можете посмотреть на моём канале:

 https://www.youtube.com/watch?v=A1b1Nr3o0cQ

ГЛАВА 7.1. QUERYBUILDER

Мы узнали, что модель - это зеркало таблицы. Так вот, чтобы строить sql запросы в рамках модели, есть инструмент, который называется QueryBuilder. Взаимодействовать с ним крайне просто. Давайте рассмотрим пример:

```
$users = User::query()->where('active', true)
    ->where('banned', false);
```

Это не что иное, как sql запрос вида:

```
SELECT * FROM users WHERE active = 1 AND banned = 0;
```

Важно понимать, что это только sql запрос. А вот исполняется он после определенных методов (`get()` , `first()` и их разновидностей):

```
$user = User::query()->where('active', true)
    ->where('banned', false)->first();
```

И вот мы исполнили запрос и получили в результате запроса одну модель. Давайте рассмотрим вариант посложнее:

```
$users = User::query()->where('active', true)
    ->where('banned', false)->get();
```

Тут мы также исполнили запрос, но уже получили в результате коллекцию - скажем так массив со всеми `User` моделями. Что же такое коллекции?

ГЛАВА 7.2. COLLECTIONS

Collections - это удобная обертка над массивом элементов, которая содержит в себе ряд полезных методов, с помощью которых мы можем взаимодействовать с её элементами:

```php
$users->sortBy('name')->filter(fn($user) => $user->id > 1);
```

Что же у нас в качестве элементов массива? Имейте ввиду, наш результат уже не имеет ничего общего с sql запросами, так как мы работаем с результатом их выполнения - с коллекцией моделей.

Что будем делать дальше? К примеру мы можем отсортировать или отфильтровать результат, используя стандартные нативные php функции - `array_map`, `array_filter`, `sort` только в рамках fluent интерфейса для нашего с вами удобства. Согласитесь это прекрасно!

> *Fluent interface (текучий интерфейс)— способ реализации в разработке программного обеспечения, объектно-ориентированного API, нацеленный на повышение читабельности исходного кода программы*

Более подробно изучить где в Laravel QueryBuilder, где модель, а где коллекция, и как все это работает, можно в ролике на моём канале:

https://www.youtube.com/watch?v=o94CesRFMuY

ГЛАВА 8. ОТНОШЕНИЯ

> *«Страус-программист каждый раз пишет одинаковые join запросы, раздувая и усложняя восприятие кода, а Laravel-разработчик выносит запросы в методы отношений и получает всю мощь Eloquent ORM»*

Eloquent ORM - это специальный инструмент программирования, который позволяет взаимодействовать с базой данных в веб-приложениях проще и удобнее. Он позволяет разработчикам использовать объекты и методы программирования для работы с данными, вместо написания сложных SQL-запросов. Это делает код более читаемым и облегчает работу с информацией в базе данных.

Вы уже узнали из предыдущей главы, что модель это отражение таблицы в базе данных. Но что, если нам нужно связать друг с другом записи в разных таблицах? Такой функционал уже заложен в моделях. Они могут реализовать различные типы отношений между записями в таблицах.

Я думаю вы уже прекрасно знаете, какие бывают связи в базе данных, но на всякий случай напомню:

- один ко многим

- один к одному

- многие ко многим

Те же самые отношения заложены и в моделях.

Давайте рассмотрим с вами все виды связей/отношений и как они реализуются на уровне моделей. Если у вас нет проблем с английским, то разобраться будет еще проще.

ГЛАВА 8.1. BELONGSTO - ОТНОШЕНИЕ “ОДИН К ОДНОМУ”

Самое простое отношение и поэтому мы начинаем именно с него.

Пример - у нас есть таблица `phones` с телефонами пользователей, в которой есть поле `user_id`. Оно помогает нам понять какому именно пользователю принадлежит данный номер телефона, и ссылается на поле `id` таблицы `users`.

Звучит как простейшая связь в базе данных. И на уровне Laravel подобное отношение будет выглядеть не сложнее.

Для начала нам потребуется модель `Phone`, которая отвечает за таблицу `phones`. Давайте создадим её с помощью artisan:

```
php artisan make:model Phone
```

Откроем её и укажем отношение, которое будет выглядеть следующим образом:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

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

Для того, чтобы указать отношение, необходимо в модель добавить соответствующий метод, который вернет экземпляр нужного отношения. Нам нужно отношение один к одному и за него в Laravel отвечает класс `BelongsTo`. А чтобы его вернуть, в моделях есть удобный метод `belongsTo`, в который нужно передать ссылку на модель, с которой мы организовываем связь, то есть ту модель, которую мы будем получать как результат обращения к отношению. У нас модель `phones` и поле `user_id`, которое ссылается на `users > id`, потому мы и указали модель `User`, которая ответственна за таблицу `users`.

Выглядит шикарно и очень лаконично, и, при условии что мы соблюдаем конвенцию наименований, нам больше не нужно ничего указывать.

Давайте также рассмотрим случай, когда конвенция наименований не соблюдается. Допустим, что база данных у нас уже была, переделывать ее нет возможности и названия таблиц и полей в них отличаются от желаемого и конвенция наименований не выполнена. Не огорчайтесь, Laravel гибкий и это не проблема:

```
return $this->belongsTo(User::class, 'foreign_key',
'owner_key');
```

Как видите, появились дополнительные параметры у метода отношения, через который мы можем задать названия связующих ключей.

Как пример могло бы быть вот так:

```
return $this->belongsTo(User::class, 'telefon_uuid',
'uuid');
```

Как удобно и просто! Далее вы можете в рамках модели обращаться к этому отношению подобно тому, как вы обращались просто к полям:

```
$phone->user->value;
```

В данном случае мы обращаемся не к методу `user()`, а именно к свойству, и получаем модель `User` со всеми ее полями.

А если мы обратимся все же как к методу `$phone->user()`, то сможем строить запрос и использовать QueryBuilder, но в наш sql запрос уже будет включено условие `where phones.user_id = users.id`, что даст нам удобство при дальнейшем взаимодействии.

Но самое интересное и волнующее новичков - это как сохранять записи в рамках отношений!? Смотрим.

В целом в рамках `BelongsTo` можно совсем не напрягаться и просто менять значение свойства `user_id`:

```
$phone->user_id = 1;
$phone->save();
```

И ничего плохого в этом не будет. Значение поля будет меняться и соответственно связь уже будет с другим пользователем, но существуют специальные методы `associate/dissociate`, чтобы привязать значение на основе модели:

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

В данном примере нам даже не нужно думать о том, что поле в базе называется `user_id`. Laravel все проставит за нас.

Опять же, все зависит от ситуации и контекста. В данном примере мы сделали тоже самое, что и в примере выше. Установили, что `$phone->user_id = $user->id`, только сработали от отношения и метода `associate`.

Что касается метода `dissociate`, то результат работы этого выражения будет следующим: `$phone->user_id = null`.

```
$phone->user()->dissociate();
```

Итак, мы знаем что `$phone->user()` у нас вернет QueryBuilder, который отвечает за построение запроса. И знаем, как просто можно менять значение через свойство и через QueryBuilder методы `associate/dissociate`! Но раз уж у нас QueryBuilder, то мы можем не только присваивать значение, но и добавлять новые записи в связующую таблицу, редактировать их и удалять.

В Laravel QueryBuilder существует несколько методов для добавления записи.

Давайте начнем с метода `save`:

```
$phone->user()->save(new User(['name' => 'CutCode']));
```

`Save` ожидает eloquent модель, которой и является `User`. Тем самым мы сохраним в таблицу `users` пользователя с именем `'CutCode'`, а также сразу автоматически укажем, что `user_id` у телефона тот, который был в объекте `$user` или созданной записи пользователя.

Есть и метод попроще, который вместо eloquent модели принимает простой массив. И называется этот метод `create`. И собственно в этом и разница между `save` и `create`. Код будет выглядеть вот так:

```
$phone->user()->create(['name' => 'CutCode']);
```

> *Eloquent Model - это основной элемент Eloquent ORM. Это класс, который связывает таблицу в базе данных с объектом в коде. Проще говоря, Eloquent Model представляет собой способ создания, обновления, чтения и удаления данных в базе данных с использованием объектно-ориентированного подхода.*

Каждая модель соответствует определенной таблице в базе данных.

В зависимости от исходных данных, вы можете выбирать между этими методами. Если уже есть модель, то сохраняете с использованием метода save, а если работаете с массивом - используйте `create`.

А что если потребуется отредактировать поля у пользователя через отношения? Не проблема, есть метод `update`:

```
$phone->user()->update(['name' => 'Oleg']);
```

Ну или снова `save`:

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

Ой, а удаление то какое сложное). Конечно же это шутка, смотрите сами:

```
$phone->user()->delete();
```

Но при этом в базе у таблицы `phones` для поля `user_id` должна быть возможность указания значения `null`.

ГЛАВА 8.2. HASMANY - ОТНОШЕНИЕ “ОДИН КО МНОГИМ”

Давайте сразу начнем с примера. У нас есть статьи (модель `Post` и таблица `posts`), есть комментарии к статьям (модель `Comment` и таблица `comments`).

Комментарии связаны со статьями через поле `post_id` (тут сразу вспоминаем отношение `BelongsTo` которое можно сразу указать на уровне модели `Comment`).

Задача - получить комментарии к определенной статье. В итоге наше отношение в модели `Post` будет выглядеть вот так:

```php
public function comments()
{
    return $this->hasMany(Comment::class);
}
```

Нас с вами здесь уже ничем не удивишь. Здесь просто изменился метод - `hasMany`, который вернет экземпляр отношения `HasMany`!

Когда мы с вами будем обращаться к `$post->comments`, то получим коллекцию с моделями `Comment` - комментариями, привязанными к статье по полю `post_id`.

А если у модели `Comment` мы указали `belongsTo` к статье:

```php
public function post()
{
    return $this->belongsTo(Post::class);
}
```

то и на уровне комментария сможем обратиться к статье `$comment->post` и получить модель `Post ($comment->post->title)`.

P.s. сразу обращайте особое внимание на то, что мы получаем иногда коллекцию, иногда модель, а иногда QueryBuilder! Очень часто новички не до конца понимают, что происходит, и именно поэтому я делаю на это акцент.

```php
// Получаем модель и нам доступен QueryBuilder
$comment->post

// Получаем Collection и QueryBuilder нам не доступен
$post->comments
```

Самые любопытные могли задуматься - почему post вернул модель, но нам доступен QueryBuilder? Это не опечатка. Модель сразу содержит в себе и QueryBuilder. Да, модели это каша отственностей,

но все они около базы данных.

Ну а тут дело дошло до добавления/редактирования/удаления записей и знаете что? Оно ничем не отличается от отношения `BelongsTo`. Только теперь у нас нет `associate/dissociate`, но оно и понятно - ведь в таблице `posts` нет поля `comment_id`, здесь связующее поле находится в `comments` - `post_id` и вот оно уже у нас `BelongsTo` с методами `associate/dissociate`!

За добавление/редактирование/удаление отвечают все те же `save/create/update/delete`.

Теперь рассмотрим методы которые сохраняют сразу несколько записей - `saveMany` и `createMany`:

```
$phone->user()->saveMany([
    new User(['name' => 'CutCode']),
    new User(['name' => 'Ivan'])
]);
```

Как видите, у нас массив моделей.

```
$phone->user()->createMany([
    ['name' => 'CutCode'],
    ['name' => 'Ivan']
]);
```

Ну а здесь массив с массивами. Думаю легко запомнить и не путаться.

ГЛАВА 8.3. HASONE - ОТНОШЕНИЕ “ОДИН К ОДНОМУ”

После `HasMany` понять `HasOne` проще простого. И скоро мы с вами в этом убедимся, ведь по большему счету это одно и тоже.

Допустим есть таблицы:

- `users` с пользователями сайта (модель `User`);

- `phones` с телефонами пользователей (модель `Phone`), в этой таблице phones есть поле `user_id`, которое является связующим с таблицей `users`.

И у нас задача - нужно получить телефон пользователя (мы уже рассматривали с вами такой пример при изучении `BelongsTo`, а сейчас работаем от обратного и получаем телефон пользователя).

При соблюдении конвенции наименований связь в модели будет выглядеть вот так:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

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

Как удобно и просто! Далее в рамках модели чтобы обратиться к результату отношения и получить модель `Phone`, мы все также, как и с `BelongsTo`, просто обращаемся к свойству, равному названию метода - `phone`:

```php
$user->phone->value;
```

Тем самым в `$user->phone` у нас уже будет модель `Phone`.

А если мы обратимся к `$user->phone()`, то сможем строить запрос и использовать QueryBuilder, но в наш sql запрос уже будет включено условие `where phones.user_id = users.id`.

Я думаю многим из вас стало скучно, ведь пока что в этом отношении ничего нового вы не увидели. Обращение такое же, как и в `belongsTo`, может добавление, редактирование и удаление изменится? Друзья, и снова нет, так как `HasOne` это полный дубль `HasMany`, единственное отличие заключается в том, что sql запрос на это отношение будет включать `LIMIT 1`.

Чтобы разобраться еще подробнее, давайте рассмотрим тот же пример, что и в `HasMany` - статьи и комментарии. Но теперь будем использовать `HasOne`.

```php
public function comments()
{
    return $this->hasMany(Comment::class);
}

public function comment()
{
    return $this->hasOne(Comment::class);
}
```

Мы с вами только что скопировали `hasMany` метод `comments`, только изменили название на `comment` и возвращаем `hasOne`.

Sql запрос для `hasMany` будет выглядеть следующим образом:

```sql
SELECT * FROM comments WHERE post_id = 1;
```

А для `hasOne`:

```sql
SELECT * FROM comments WHERE post_id = 1 LIMIT 1;
```

Как я вам говорил ранее, `hasOne` это `hasMany`, но с `LIMIT`. И я думаю вы уже догадались, что методы queryBuilder (для добавления, редактирования, удаления и т.д.) для `hasOne` ничем не будут отличаться от `hasMany`. Все тоже самое!

ГЛАВА 8.4. РАСШИРЕННОЕ ИСПОЛЬЗОВАНИЕ HASONE

Друзья, прежде чем переходить к следующему типу отношений, давайте рассмотрим еще один вариант применения `HasOne`. Часто бывает так, что у модели есть много записей, но нужно получить только одну, в рамках отношения `hasOne`. К примеру, нужно просто показать последнюю запись.

Предположим, что у нас есть `User` и множество его комментариев (`Comment`), все комментарии привязаны к `User` по полю в таблице `user_id`. Мы строим отношение `HasOne`, но при этом дополнительно указываем метод `latestOfMany`:

```
public function lastComment(): HasOne
{
    return $this->hasOne(Comment::class)->latestOfMany();
}
```

Тем самым мы возьмем запись у которой `id` будет последним (максимальным).

Обратный метод, с самой старой записью - `oldestOfMany`:

```
public function firstComment(): HasOne
{
    return $this->hasOne(Comment::class)->oldestOfMany();
}
```

По-умолчанию поле таблицы `id`, но мы можем передавать и отличное от `id` первым аргументом (`'identity'`):

```
public function firstComment(): HasOne
{
    return $this->hasOne(Comment::class)
        ->oldestOfMany('identity');
}
```

Есть и более гибкий метод `ofMany`, где мы можем указывать сразу несколько полей и агрегатных функций по ним, а также добавить запрос для отношения:

```
public function currentPricing()
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function ($query) {
         $query->where('published_at', '<', now());
    });
}
```

Это тот же самый `HasMany` с `LIMIT 1`, но с дополнительным условием сортировки. Теперь мы готовы перейти к более сложным отношениям, по крайней мере они такими кажутся на первый взгляд. На самом деле всё не так страшно, вот увидите.

P.S. В процессе написания этого гайда появился новый подход для объявления отношения `HasOne` и он может многим из вас понравится. Я и сам буду им пользоваться!

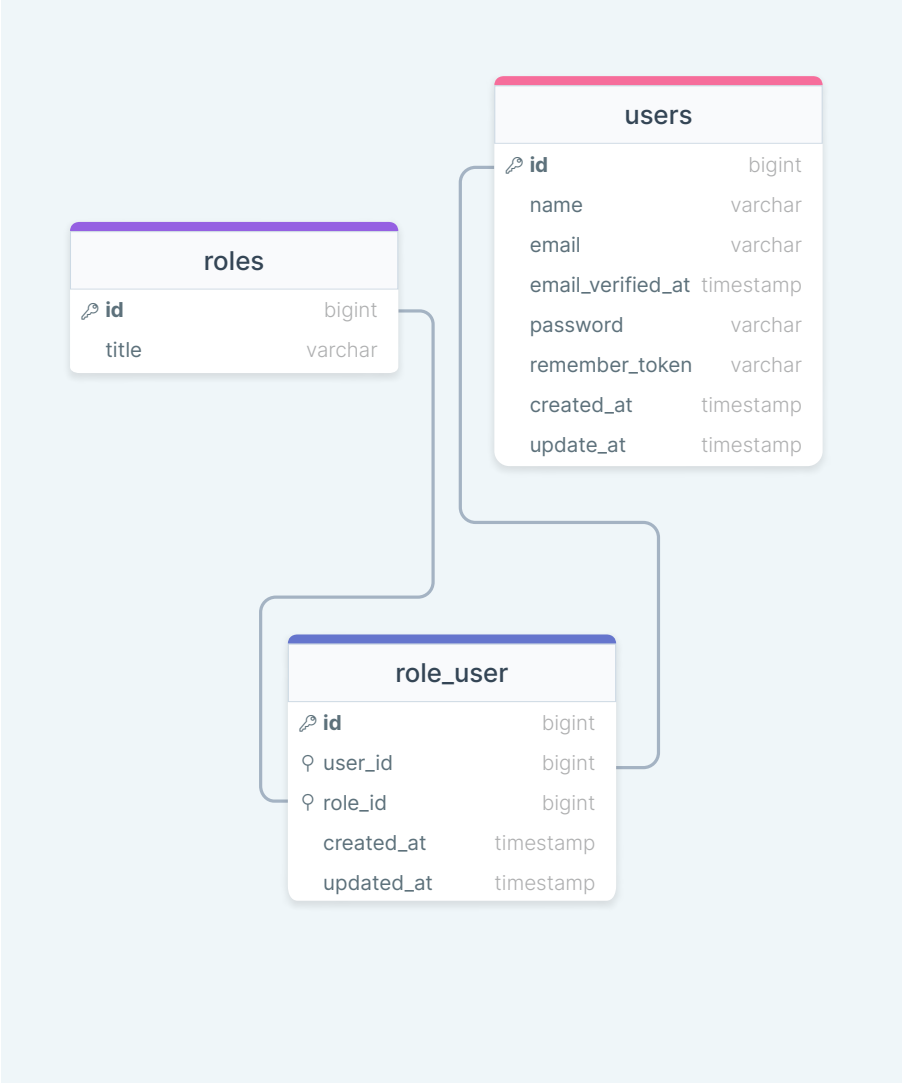
Выглядит он следующим образом:

```
public function firstComment(): HasOne
{
    return $this->comments()->one()->oldestOfMany();
}
```

Это еще раз подчеркивает, что `HasOne` ничем не отличается от `HasMany`. Обратите внимание - мы уже не возвращаем напрямую экземпляр отношения `HasOne` через метод `hasOne`. Вместо этого мы взяли уже объявленное отношение comments (`HasMany`) и просто трансформировали его в `HasOne` добавив метод `one()`. На мой взгляд супер круто.

ГЛАВА 8.5. BELONGSTOMANY - ОТНОШЕНИЕ МНОГИЕ КО МНОГИМ

Предположим, что у нас все тот же пользователь `User` и у него несколько ролей для прав доступа. Он у нас и модератор и автор статей, а еще и администратор. Роли у нас уже прописаны в базе данных в таблице `roles` и модель, ответственная за них, соответственно `Role` (ведь мы не хулиганим с конвенцией наименований).



Если бы мы были с вами страус-программистами, то хранили бы роли скажем, в строковом поле. Так у `User` было бы поле `role`, где мы бы хранили все значения ролей пользователя через запятую (1,2,3), и постоянно бы сталкивались со сложностью поиска наличия ролей, сортировки по ним и внесению изменений. А если бы нам понадобилось добавить еще и параметры к ролям, то это вообще боль, крайне грязный подход. CutCode против такого положения вещей!

Мы с вами создадим таблицу с использованием конвенции наименований, `pivot` таблицу. Мы ее обсуждали в разделе по конвенции наименований и называться она у нас будет `role_user`. Это связующая таблица, которая связывает `users` и `roles` с их идентификаторами. Напомню, что название `pivot` таблицы `role_user` получилось на основе названий двух таблиц `users` и `roles` и указали имя именно `role_user` по конвенции наименований - обе таблицы в единственном числе, в порядке алфавита `r` раньше чем `u`, и поэтому мы начинаем с `role`.

Вот так выглядит структура БД:

```
users
   id - integer
   name - string

roles
   id - integer
   name - string

role_user
   user_id - integer
   role_id - integer
```

А вот так миграция:

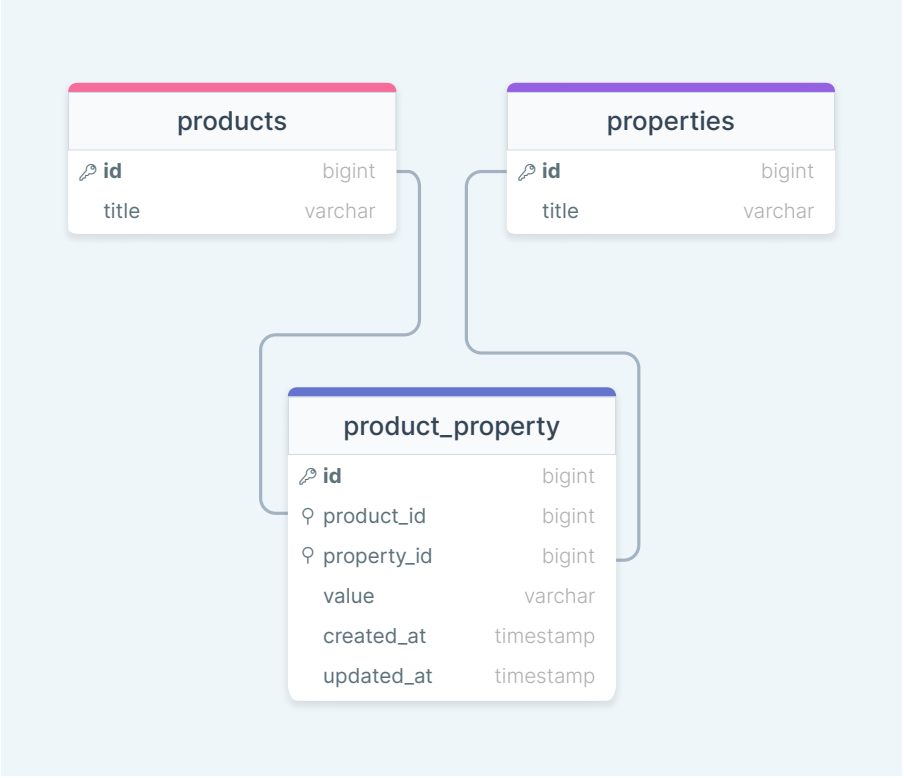
```
Schema::create('role_user', function (Blueprint $table) {
    $table->id();

    $table->foreignId('user_id')
        ->constrained()
        ->cascadeOnDelete()
        ->cascadeOnUpdate();

    $table->foreignId('role_id')
        ->constrained()
        ->cascadeOnDelete()
        ->cascadeOnUpdate();

    $table->timestamps();
});
```

Ну и дело дошло до связи:

```
public function roles()
{
    return $this->belongsToMany(Role::class);
}
```

Согласитесь крайне элегантно.

Теперь для того, чтобы мы могли получить все роли юзера, достаточно обратиться к свойству `$user->roles` и в результате у нас будет не `id`, а коллекция со всеми моделями `Role`, привязанными к конкретному пользователю.



Теперь представим ситуацию: у вас есть товар (модель `Product`, таблица `products`), который через `belongsToMany` связан с характеристиками товара (модель `Property` таблица `properties`). Но вы хотите не просто указать какая характеристика привязана к товару, а еще и указать её значение! Скажем, есть товар - телефон Samsung, и у него характеристика Оперативная память. Отлично, есть такая, но а сколько именно этой памяти? На помощь приходят так называемые `pivot` значения! По структуре выглядит это так:

```
Schema::create('product_property', function
(Blueprint $table) {
    $table->id();

    $table->foreignId('product_id')
        ->constrained()
        ->cascadeOnDelete()
        ->cascadeOnUpdate();

    $table->foreignId('property_id')
        ->constrained()
        ->cascadeOnDelete()
        ->cascadeOnUpdate();

    $table->string('value');

    $table->timestamps();
});
```

Мы просто добавили текстовое поле со значением. Но вот незадача: когда мы обращаемся к `$product->properties`, мы получаем коллекцию с моделями `Property` и ничего не знаем о `value`! Но исправляется это просто, на уровне объявления отношения:

```
public function properties()
{
    return $this->belongsToMany(Property::class)
        ->withPivot('value');
}
```

Тем самым мы указали, что нам нужно возвращать не только модель, но и поле `value`. Теперь мы можем сделать так и получить результат:

```
foreach ($product->properties as $property) {
    echo $property->pivot->value;
}
```

Мы обращаемся через свойство `pivot` (это ключевое слово можно менять, но вы об этом узнаете, как только пройдете курс молодого бойца Laravel, и переключитесь на официальную документацию).

Друзья, чуть было не забыл о том, как сохранять и изменять привязанные записи (модели).

Для этого есть методы `attach/detach` и `sync/toggle`:

```
$product = Product::find(1);

$product->properties()->attach(1);
```

Так мы привязали к товару характеристику `id 1`.

```
$product->properties()->attach(1, ['value' => '128 mb']);
```

Так мы привязали к товару характеристику `id 1` и сохранили pivot значение `value = 128 mb`.

```
$product = Product::find(1);

$product->properties()->detach(1);
```

Я думаю вы и так догадались, что здесь мы удалили характеристику `id 1` из списка.

Да, друзья, у этого вида отношений много удобных для нас методов. Давайте рассмотрим еще один, `sync`:

```
$product->properties()->sync([1,2,3]);
```

Мы сделали следующее: привязали к товару характеристики с `id` 1, 2, 3, а вот все остальные, которые уже были привязаны до этого, были удалены! То есть, если до этого у товара были характеристики 1, 4, 5, 6, то после остались только 1, 2, 3.

Далее на очереди метод `toggle`:

```
$product->properties()->toggle([1, 2, 3]);
```

В данном случае мы не просто добавили 1, 2, 3, а переключили их. Это значит, что если у товара уже была характеристика, скажем 1, то она удалится. При помощи `toggle` мы либо используем характеристику, или нет.

Рассмотрим на примере.

```
// начальное состояние
Характеристик товара []
Характеристики id 1, 10, 20

//добавляем характеристику id 10
$product->toggle([10])
Характеристик товара [10]

//переключаем характеристику id 10 и добавляем
характеристику id 20
$product->toggle([10, 20])
Характеристик товара [20]
```

И в рамках `sync` и в рамках `toggle` мы также можем работать и с `pivot` значениями:

```
$product->properties()->sync([1 => ['value' => '128 mb'],
2, 3]);
```

ВНИМАНИЕ!!! Частый вопрос от новичков - когда использовать `belongsToMany`, а когда `HasMany`. Простой ответ - если к сущности нужно привязать уникальные записи, каждый раз новые, например есть пользователь и он имеет квартиры, они принадлежат только ему, это уникальные сущности, то у пользователя будет `hasMany` связь квартирам. Но если у Статьи есть Категории, и те же самые категории могут быть и у других статей, то здесь Катего-

рии это полностью самостоятельная сущность, живущая отдельно от статей. Но может быть привязана к разным сущностям, тогда мы используем `belongsToMany`, но, как в примере с характеристиками, можем через pivot поля добавлять уникальность.

Еще хороший пример - услуги, которые оказывает исполнитель. Услуги располагаются в отдельной таблице в базе данных, и в целом существуют и без исполнителей. Но мы также можем привязать их к исполнителю, и при этом через `pivot` поля добавить уникальные для связи значения, такие как стоимость услуги. В итоге одна услуга может быть привязана к разным исполнителям, но в рамках связи стоимость будет уникальна между ними.

ГЛАВА 8.6. HASONETHROUGH - ОТНОШЕНИЕ ОДИН К ОДНОМУ ЧЕРЕЗ ТАБЛИЦУ

Наращиваем сложность и переходим к более интересным связям - отношения через таблицу. И самое простое отношение через таблицу - это отношение один к одному. Отличный пример приведен в официальной документации, давайте им и воспользуемся.

Пример: у нас есть механики (Модель `Mechanic`, таблица `mechanics`), есть машины, которые они ремонтируют (`Car`, `cars`), а у машин есть владельцы - (`Owner`, `owners`).


mechanics
id bigint
name varchar
cars
id bigint
model varchar
mechanic_id bigint
owners
id bigint
name varchar
car_id bigint

Схема понятна и перед нами стоит задача в рамках модели `Mechanic`, обратиться к владельцу, при этом не трогая модель с машинами напрямую, а обратиться через нее.

Давайте начнем со структуры базы данных:

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

Ничего сложного.

Связь будет выглядеть следующим образом:

```
class Mechanic extends Model
{
    /**
    * Get the car's owner.
    */
    public function carOwner()
    {
        return $this->hasOneThrough(Owner::class, Car::
class);
    }
}
```

То есть мы указываем, что хотим от модели механика сразу получить владельца автомобиля, и достигаем этого через метод отношений `hasOneThrough`, где первый аргумент это модель, которую мы хотим получить в результате, а вторая - через какую таблицу (модель) мы будем ее получать.

Опять-таки, по конвенции наименований все просто - у нас есть модель `Mechanics`, значит поле, через которое мы будем искать владельца в таблице `cars`, будет `mechanic_id`, а далее в самой таблице `owners` мы уже работаем от таблицы `cars`, которую мы получили на предыдущем шаге и по `car_id` узнаем владельца.

Ну а если мы все-таки нарушаем конвенцию, то вот такая шпаргалка нам в помощь:

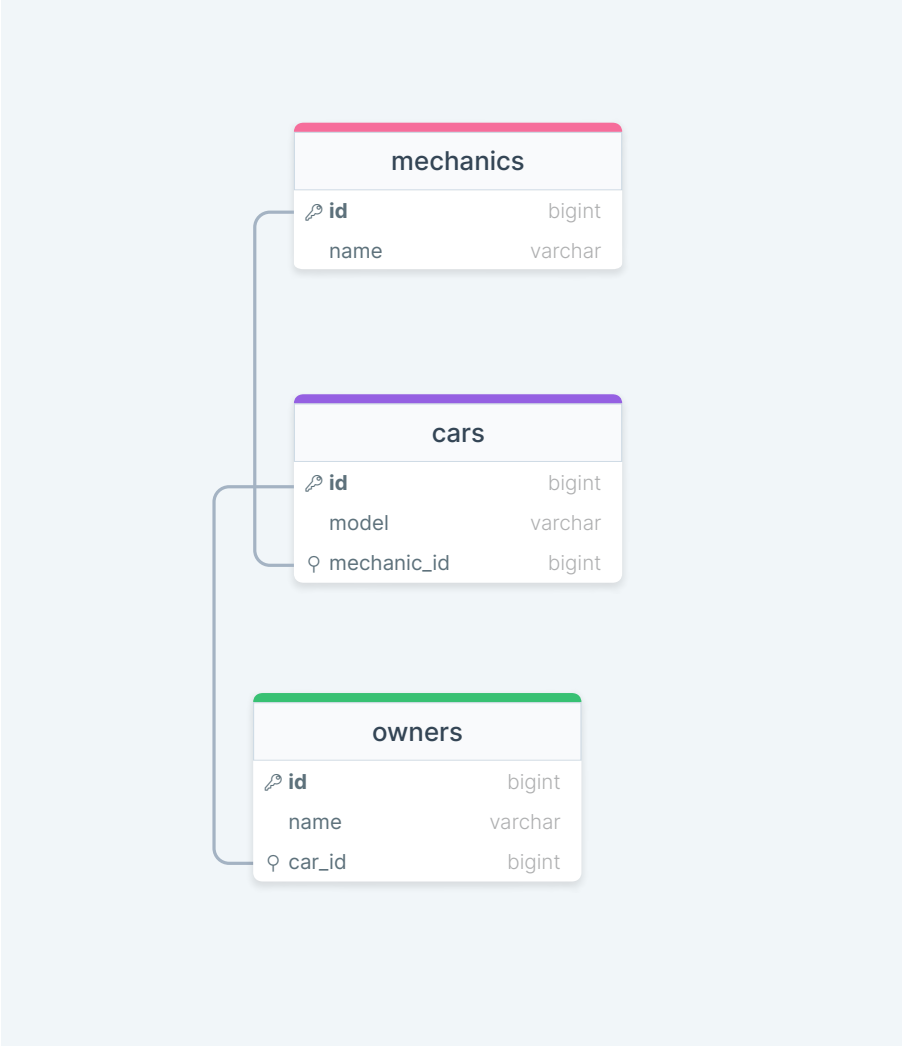
```
public function carOwner()
{
    return $this->hasOneThrough(
        Owner::class,
        Car::class,
        'mechanic_id', // Ключ в таблице cars - связь с
текущей таблицей mechanics
        'car_id', // Ключ в таблице owners связь с таблицей
cars через которую мы двигаемся
        'id', // Primary key в mechanics
        'id' // Primary key в cars
    );
}
```

Что касается добавления записей, то здесь тоже самое как и в `hasOne/HasMany`, только отсутствуют `save/saveMany/createMany`.

Остается только `create` из-за специфики отношения.

ГЛАВА 8.7. HASMANYTHROUGH - ОТНОШЕНИЕ ОДИН КО МНОГИМ ЧЕРЕЗ ТАБЛИЦУ

Друзья давайте я здесь не буду умничать и скажу вам, что принципы работы с этим видом отношения очень напоминает `hasOneThrough` - только мы получаем не одну запись, а коллекцию. Пример выше (с механиками и владельцами) подойдет полностью и по моделям и структуре, методы на добавление записей работают по тем же правилам.



Единственное отличие - другой метод отношений и выглядеть в модели он будет следующим образом:

```
class Mechanic extends Model
{
    /**
    * Get the car's owner.
    */
    public function carOwners()
    {
        return $this->hasManyThrough(Owner::class, Car::
class);
    }
}
```

P.S. В процессе написания этой книги Laravel обновился и появился новый подход объявления отношений и тоже через существующие методы отношений, подобно тому как это было с `HasOne`.

Итак встречайте новый подход:

```
class Mechanic extends Model
{
    public function carOwners()
    {
        return $this->through('cars')->has('owners');
    }
}
```

В данном случае мы путешествуем через отношение, но у нас в основе не указание модели, а указание отношения, через которое мы двигаемся и отношение которое получаем в результате.

В итоге у механика есть множество машин, через `HasMany` отношение `cars` которое мы и указали в методе `through`:

```php
class Mechanic extends Model
{
    public function cars()
    {
        return $this->hasMany(Car::class);
    }
}
```

А в `Car` модели есть отношение `owners` (`HasMany`):

```php
class Car extends Model
{
    public function owners()
    {
        return $this->hasMany(Owner::class);
    }
}
```

Тем самым мы с вами получили тот же результат, но двигались через готовые методы отношений. Ну а если брать `HasOneThrough` , то все что нам необходимо, это то, чтобы в моделе `Car` было отношение `owner` , которые у нас `HasOne` :

```php
class Car extends Model
{
    public function owner()
    {
        return $this->hasOne(Owner::class);
    }
}
```

Ну и тогда в `Mechanic` необходимо указать в `has` - `owner` :

```php
class Mechanic extends Model
{
    /**
    * Get the car's owner.
    */
    public function carOwner()
    {
        return $this->through('cars')->has('owner');
    }
}
```

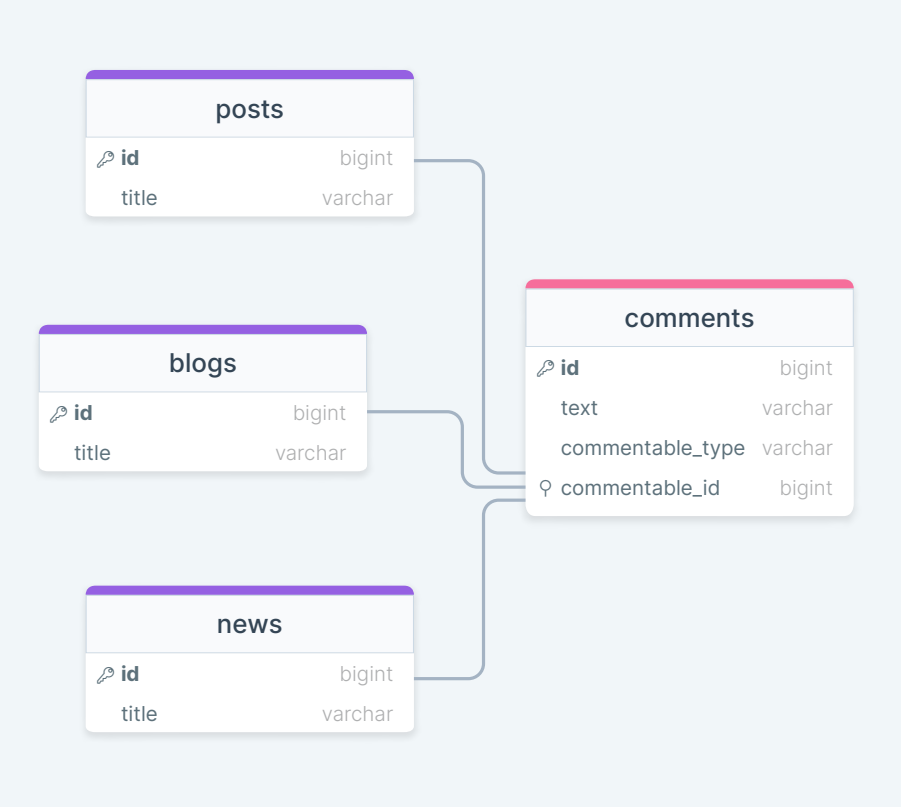
ГЛАВА 8.8. ПОЛИМОРФНЫЕ ОТНОШЕНИЯ

> *«Новички очень боятся этого раздела документации. Страус-программисты фанатично реализуют просто ужасную структуру базы данных с большим количеством дублей, не зная о всех возможностях Laravel. А я же вас обрадую и скажу, что полиморфные отношения и структура в базе данных только облегчит нам жизнь и сделает наш код гибким, а количество таблиц и моделей сократится в разы. Погнали!»*

8.8.1. MORPHTO И MORPHMANY - ПОЛИМОРФНОЕ ОТНОШЕНИЕ ОДИН К ОДНОМУ И ОДИН КО МНОГИМ

Давайте начнем с примера, который охватит сразу два вида отношений. У нас есть статьи и есть комментарии к ним. Есть таблица `posts` и модель `Post`, а также есть таблица `comments` и модель `Comment`. Пока что все можно реализовать с помощью инструментов, которые мы изучили в прошлых главах. `Post` связан с `comments` отношением `HasMany`, с его помощью мы получаем все комментарии к текущей статье. В свою очередь каждый комментарий связан отношением `BelongsTo` с `Post` (держите в голове эти классические отношения, они нам пригодятся).

Идеальный мир, пока на нашем проекте не появились новые сущности - разделы Новости и Блог. И в этих разделах тоже очень нужно выводить комментарии. И тут пользоваться нашей старой таблицей `comments` уже не получится, ведь `post_id` привязан к статьям, и мы строго соблюдаем конвенцию наименований. И как же быть?



Придется делать таблицы `blog_comments`, `post_comments`, `new_comments`? А что если появятся еще разделы на которых надо будет сделать комментарий? Ужас, не правда ли?

Но нам на помощь приходят полиморфные связи. Полиморфные - то есть имеющие множество форм, как и наши комментарии, которые могут быть и у статей, и у новостей, и у чего угодно. Они полиморфны. Давайте их реализуем.

Начнем со структуры. Будем придерживаться конвенции наименований, и здесь будет один небольшой нюанс:

```
posts
   id - integer
   title - string

blogs
   id - integer
   title - string

news
   id - integer
   title - string

comments
   id - integer
   text - text
   commentable_id - integer
   commentable_type - string
```

Как видите, у нас есть и статьи, и новости, и блог. А таблица с комментами всего одна, только она не с ключом `post_id`, а с `commentable_id`, который у нас будет ссылаться иногда на статьи, иногда на блог, а иногда на новости. А вот чтобы понять, куда именно надо ссылаться, мы и добавили строковое поле `commentable_type`, в котором будет название модели, на которую мы ссылаемся, имя класса (`App\Models\Post или App\Models\Blog` и так далее).

Как раз вот это `commentable` - это применение конвенции наименований, но в целом он может быть и другим, просто так принято у Laravel-разработчиков, когда содержится название нашей таблицы в единственном числе `comment` и приписка - `able`. Да иногда такие названия выглядят странно и грамматически не верны, но что ни сделаешь ради удобства и лаконичности.

Да, друзья, еще раз скажу - если вам не нравится такой префикс, то вы можете использовать и свой, и далее просто указать его в отношениях.

OK, со структурой разобрались. А как же будет выглядеть отношение в моделях? Итак, друзья, вспоминайте `HasMany`, а теперь смотрите на `morphMany`:

```
class Post extends Model
{
    /**
    * Get all of the post's comments.
    */
    public function comments()
    {
        return $this->morphMany(Comment::class,
'commentable');
    }
}
```

Как видите, есть сходство в наименовании. У нас `hasMany` превратился в похожий `morphMany` с одной особенностью: во втором параметре, где мы указываем тот самый префикс, мы выбрали `commentable`. И теперь Laravel знает, что работать нужно с полями `commentable_id` для определенной записи и `commentable_type` для выбора модели и соответственно таблицы.

Давайте также в рамках модели `Comment` укажем полиморфную связь один к одному, которая будет ссылаться или на `Post`, или на `News`, или на `Article`.

```
class Comment extends Model
{
    /**
    * Get the parent commentable model (post, new
    * or article).
    */
    public function commentable()
    {
        return $this->morphTo();
    }
}
```

И снова вы должны увидеть сходство. Будь у нас как раньше `post_id`, то был бы `belongsTo`. А теперь `morphTo`, только модели могут быть разными, поэтому мы не передаем привычный нам класс первым аргументом, а вот название метода как раз наш префикс с `commentable`. И если поменяли его в таблице, то и название метода нужно поменять.

Ну и давайте еще разок повторим: в данном случае мы указываем, что таблица с комментариями будет иметь отношение один к одному `BelongsTo` к определенной записи. И за счет прокаченного `MorphTo` мы через поле `commentable_type` указываем, к какой именно. `MorphTo` это тот же `BelongsTo`, с такими же методами `associate/disassociate` только прокачен до полиморфизма. С `belongsTo` у нас было бы поле скажем `post_id` с четкой привязкой к `Post`, а вот `MorphTo` имеет `commentable_id` с привязкой исходя из `commentable_type`.

Давайте взглянем на модели, у которых могут быть комментарии. Вот так будет выглядеть само отношение в наших моделях, которые обращаются к комментариям, и тут все привычно:

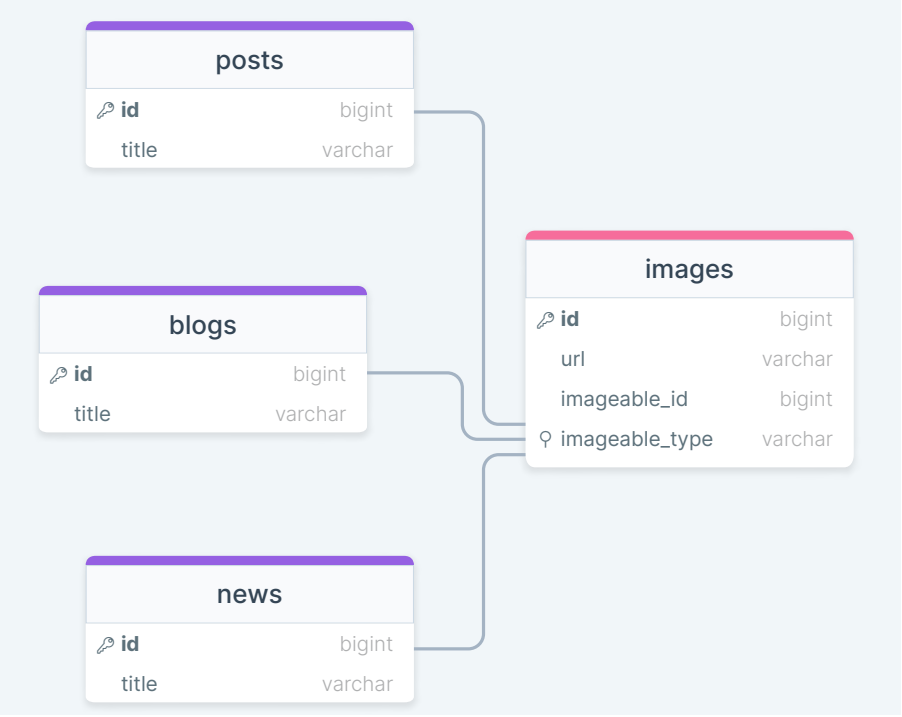
```
$post->comments
$blog->comments
$newItem->comments
```

Сохраняются записи точно также как и с `HasMany`. И в целом `morphMany` наследует `HasMany` и по логике взаимодействия идентичен.

8.8.2. MORPHONE - ПОЛИМОРФНОЕ ОТНОШЕНИЕ ОДИН К ОДНОМУ

В этом типе отношений я вас вряд ли удивлю, все как и с `MorphMany`. Только логика связи подразумевает, что запись имеет только одну запись в связи. На самом деле мы имеем дело с `HasOne`, только прокаченного до полиморфных отношений.

Пример взят из документации и он вполне нам подходит - мы имеем те же самые статьи, новости и блог. У каждой записи всех этих сущностей есть url изображения, которое хранится у нас в отдельной таблице. Кроме ссылки на картинку есть несколько мета полей. Но мы снова не хотим создавать кучу однотипных таблиц для каждой сущности, поэтому воспользуемся полиморфными отношениями.



Структура у нас следующая:

```
posts
   id - integer

blogs
   id - integer

news
   id - integer

images
   id - integer
   url - string
   imageable_id - integer
   imageable_type - string
```

Вот так выглядит отношение в модели `Post`:

```php
class Post extends Model
{
    /**
    * Get the post's image.
    */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

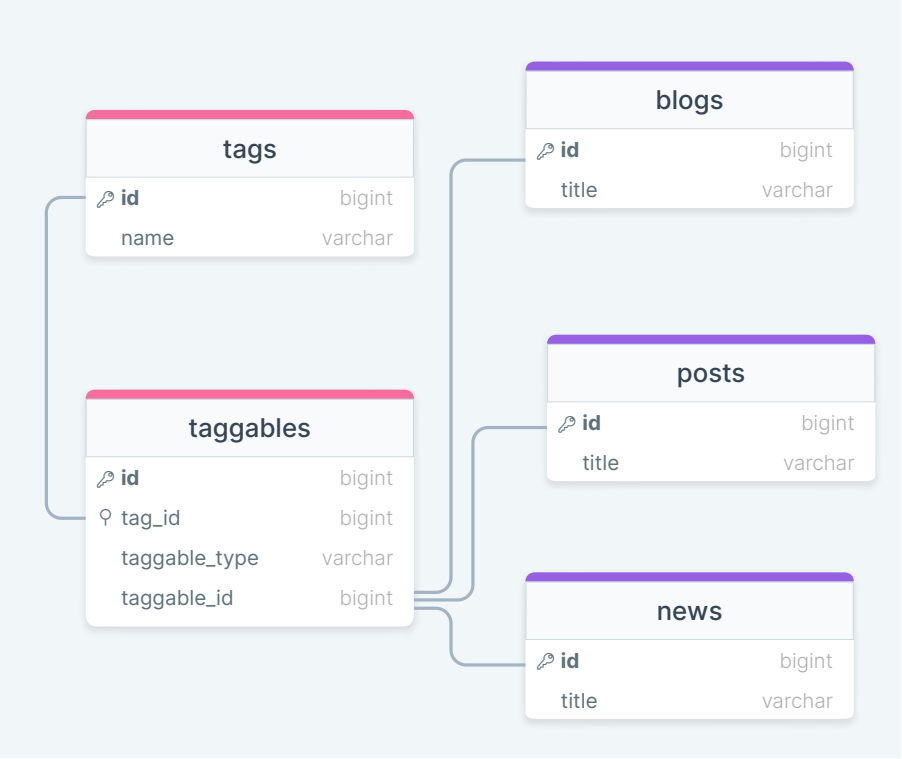
Думаю вы уже и без меня видите сходство с `morphMany`, поменялся только метод, на `MorphOne`: это все тот же `HasOne`, но прокаченный до полиморфных отношений.

P.S. Рекомендую вернуться к отношению `HasOne` чтобы освежить память и вспомнить про `latestOfMany`, `oldestOfMany` и `ofMany` методы, ведь здесь они также присутствуют. А вот что

касается добавления записей, то тут все тоже самое как и в `HasOne`. Да и новый подход через метод `one()` также доступен.

8.8.3. MORPHTOMANY/MORPHEDBYMANY - ПОЛИМОРФНОЕ ОТНОШЕНИЕ МНОГИЕ КО МНОГИМ

Друзья, сейчас рекомендую на всякий случай освежить память и вспомнить, что такое `BelongsToMany` отношение и пример который мы разбирали. Представьте, что у нас есть все те же самые статьи, новости и блоги. У каждой сущности есть теги, и если бы теги были не полиморфные и были только у статей, то мы бы просто сделали таблицу `tags`, модели `Tag` и `Post`. И связали всё это `BelongsToMany` связью и не знали бы бед. Но снова задача усложнилась и появились новости и блог, а вместе с ними и теги для каждой сущности. И снова к нам на помощь приходят полиморфные связи.



Итак, теги теперь у нас везде и используются для всех сущностей. А вот статьи, новости и блог полиморфны для тегов.

Структура базы следующая:

```
posts
   id - integer
   title - string

blogs
   id - integer
   title - string

news
   id - integer
   title - string

tags
   id - integer
   name - string

taggables
   tag_id - integer
   taggable_id - integer
   taggable_type - string
```

Как видите, таблица `taggables` это то, что у нас было бы, если бы мы делали `blog_tag` с использованием простого `belongsToMany`. Задача у нас сложнее и мы ее немного прокачали.

Тут уже все меняется для моделей и если говорить о модели `Tag`:

```
class Tag extends Model
{
    /**
    * Get all of the posts that are assigned this tag.
    */
    public function posts()
    {
        return $this->morphedByMany(Post::class,
'taggable');
    }

    /**
    * Get all of the blogs that are assigned this tag.
    */
    public function blogs()
    {
        return $this->morphedByMany(Blog::class,
'taggable');
    }
}
```

То есть нам нужно указать все связующие модели с помощью метода `morphedByMany`. А в каждой модели у которой есть теги делаем одинаковый метод:

```
class Post extends Model
{
    /**
    * Get all of the tags for the post.
    */
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

Можно даже вынести в `trait` и просто вешать на все модели которые подразумевают наличие тегов. А получать и добавлять записи также как и с `BelongsToMany`.

ГЛАВА 8.9. EAGER LOAD

А вы слышали о проблеме N+1? Если нет, то обязательно услышите. А прочитав этот раздел, она Вас не напугает, потому что у вас на вооружении Laravel и QueryBuilder.

Итак, давайте начнем с самой проблемы, и лучше всего разобраться в вопроса нам поможет простой пример.

Давайте все еще обсуждать статьи и комментарии. У каждого комментария должен быть автор, правильно? Вот есть таблица `comments` и в ней `author_id`, которое отношением `BelongsTo` ссылается к таблиице `authors (Author)`. Вы отлично организовали структуру и отношения, и начинаете писать логику. Вы взяли все комментарии у поста `$post->comments`, решили пройтись по каждому и обратиться к автору, чтобы например вывести в шаблоне имя автора:

```
foreach($post->comments as $comment) {
    echo $comment->author->name;
}
```

Все будет работать. И если комментариев не много, то вы возможно даже не заметите время на выполнение запроса. Особенно если не пользуетесь дебагбаром и не отслеживаете все sql запросы (за что я бы вас поругал - настоятельно рекомендую всегда использовать дебагбар и следить за запросами к БД, ведь для веб приложений это основа производительности).

Так вот, все работает, но знаете сколько запросов вы отправите к базе данных? Сперва первый на все комменты, вот здесь `$post->comments`. Потом по каждому комментарию еще дополнительный запрос вот здесь `$comment->author->name`. И если у поста 100 комментов, то это 101 запрос, а мы еще пост получали - это уже 102 запроса, и это ужас!

Но мы можем взять всех авторов в одном запросе, до вызова

`comments`, и сохранить в памяти. А потом при обращении к `$comment->author` уже не отправлять еще запросы, а брать данные из памяти.

Как? С помощью возможностей Eager Load и метода `with`!

Просто при получении `comments` указываем, что нам также потребуется подгрузить заранее `author`. И, друзья, помните, что `with` это не `join`. `with` это дополнительный запрос на основе родительского, в котором мы получаем все дочерние элементы отношения заранее.

Выглядит это следующий образом:

```
$comments = Comment::query()->with('author')->get();
```

либо:

```
$post = Post::query()->with('comments.author')->
    where('id', 1)->first();
```

либо если мы уже имеем модель `Post` (например, через `route binding`) и нам надо добавить Eager Load, то пользуемся методом `load`.

Как раз это легче понимать, зная что Eager Load это не `join`, а просто запрос, который сохранит результат в памяти и его можно реализовать в любой момент перед вызовом самого отношения:

```
$post->load('comments.author');
```

В процессе локальной разработки Laravel вас будет подстраховывать, и подскажет, если вы забудете задать eager load. Настроить это крайне просто, зато пользу ощутите сразу.

Итак, в AppServiceProvider добавьте правило:

```
public function boot(): void
{
    Model::shouldBeStrict(!app()->isProduction());
}
```

Теперь, если мы с вами забудем про Eager Load, то будем получать Exception с ошибкой. Поэтому обязательно добавляем флаг `!app()->isProduction()`, так как на продакшене нам такие ошибки не нужны в любом случае.

`shouldBeStrict` также будет для нас помощником в работе с магией Eloquent. Если без `shouldBeStrict` мы обратимся к несуществующему атрибуту модели, то просто получим `null` и даже не будем подозревать об этом.

```
$model->foo // null
```

А вот с `shouldBeStrict` сразу получим ошибку. И бонусом - если будем наполнять значения модели, не указанные в свойстве `$fillable`, то также получим ошибку.

ГЛАВА 8.10. QUERYBUILDER ДЛЯ ОТНОШЕНИЙ

Имея отношения в рамках QueryBuilder, мы также можем строить удобные условия, для этого есть ряд методов. Давайте их рассмотрим.

`has` - получим все статьи у которых есть хотя бы 1 комментарий:

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

Или чуть сложнее - хотя бы один комментарий, который содержит в тексте слово `code`, ну если не усложнять, то `has` с условием):

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

Есть и обратные: `doesntHave` и `whereDoesntHave`. Тоже самое только когда нет ни одной записи.

А вот еще классный и простой метод для получения записей статей у которых комментарии соответствуют условию:

```
$posts = Post::whereRelation('comments', 'is_approved',
false)->get();
```

В этом примере получим все статьи у которых комменты не подтверждены.

ГЛАВА 8.11. АГРЕГАТНЫЕ ФУНКЦИИ ДЛЯ ОТНОШЕНИЙ

Друзья, вам часто будет необходимо выполнять задачу по простому получению записей из таблицы - количество всех записей, сумму значений по какому-то полю или среднее значение. Для этого применяются агрегатные функции.

Агрегатная функция выполняет вычисление над набором значений и возвращает одно значение. В табличной модели данных это значит, что функция берет ноль, одну или несколько строк для какой-то колонки и возвращает единственное значение.

Например, у вас листинг статей. И вам не нужно брать все комментарии и делать `$post->comments->count()`, ведь тем самым мы грузим в память большое число моделей, а нам нужен примитивный integer.

Но и здесь в Laravel есть удобные методы в рамках Query Builder.

Давайте начнем с количества:

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

Тем самым мы взяли только количество комментариев и обратиться к ним можно через свойство `comments_count` (название таблицы и суффикс `_count`):

```
foreach ($posts as $post) {
    echo $post->comments_count;
}
```

Таким же образом работают и другие агрегатные функции. Например с помощью методов `withSum`, `withMin`, `withMax`, `withAvg`. Только здесь шаблон свойства будет `{таблица связи}_{агрегатор функция}_{название поля}`:

```php
$posts = Post::withMax('comments', 'likes')->get();

foreach ($posts as $post) {
    echo $post->comments_max_likes;
}

$posts = Post::withAvg('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_avg_votes;
}

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

ГЛАВА 9. MUTATORS/ACCESSORS/CASTS

 https://laravel.com/docs/eloquent-mutators

ГЛАВА 9.1. MUTATORS/ACCESSORS

В главе про модели я рассказал, что обращаясь к свойствам, мы тем самым получаем значение из таблицы. А когда присваиваем значение свойствам, то мы в дальнейшем сохраняем их, создавая новые записи, или обновляем существующие. Модель - это «обертка» над таблицей, и когда мы делаем обращение к свойству `$user->name`, то мы берем значение этого поля в таблице. А если делаем присваивание `$user->name = 'Anton'`, и потом `$user->save`, то мы либо сохраним новое значение в таблице, либо добавим новую запись если её еще не было.

Так в Laravel работает магия PHP, а именно магические методы `__get` и `__set` при обращении к свойствам и при присваивании им значений соответственно. В эти процессы мы с вами можем вмешаться и менять получаемые значения.

Давайте представим, что у нас есть модель `User`, и у него есть поле `phone` (телефон), которое в базе хранится как `integer` вида 79999999999. А нам на сайте нужно вывести его с плюсиком +79999999999 (я уже молчу о более полезных и сложных трансформациях типа конвертации дат и сумм под локализацию пользователя).

В модели мы можем объявить `Accessor` для поля и выглядеть он будет следующим образом:

```php
class User extends Model
{
    /**
    * Get the user's first name.
    *
    * @return \Illuminate\Database\Eloquent\Casts\
Attribute
    */
    protected function phone(): Attribute
    {
        return Attribute::make(
            get: fn ($value) => '+' . $value,
        );
    }
}
```

И в момент, когда мы обратимся к свойству `phone`, у модели `$user->phone` сработает этот метод и добавит плюсик к тому значению что уже есть в базе.

А также мы можем добавлять `Accesors` даже к полям, которых нет в базе, то есть создавать новые поля. Скажем, у пользователя есть имя и фамилия, а мы хотим выводить на сайте и то и другое, и делать это с использованием только одного свойства. Не проблема:

```
class User extends Model
{
    /**
    * Get the user's first name.
    *
    * @return \Illuminate\Database\Eloquent\Casts\
Attribute
    */
    protected function name(): Attribute
    {
        return Attribute::make(
            get: fn () => $this->first_name . ' ' . $this
->last_name,
        );
    }
}
```

И в итоге можем обращаться к свойству `$user->name` и получим фамилию и имя.

То же самое можно сделать в момент присвоения значения и в итоге сохранять трансформированное значение. Например, тот же телефон может приходить с формы регистрации в разном формате, а нам нужно привести его к `integer` и убрать все лишнее:

```
class User extends Model
{
    /**
    * Get the user's first name.
    *
    * @return \Illuminate\Database\Eloquent\Casts\Attribute
    */
    protected function phone(): Attribute
    {
        return Attribute::make(
            set: fn ($value) =>
trim(preg_replace('/^1|\D/', "", $value)),
        );
    }
}
```

ну и само собой, мы можем объединить и `get` и `set` в одном методе:

```
return Attribute::make(
    get: fn ($value) => '+' . $value,
    set: fn ($value) => trim(preg_replace('/^1|\D/', "",
$value)),
);
```

ГЛАВА 9.2. CASTS

Только что мы вооружились знаниями по Mutators/Accesors и теперь можем легко понять `Casts`, ведь в их основе как раз и лежат Mutators/Accesors.

Во-первых, есть уже готовые варианты в Laravel и добавить их можно с помощью свойства `$casts` в рамках модели. Приведем тип при получении и сохранении к булеан.

```
class User extends Model
{
    /**
    * The attributes that should be cast.
    *
    * @var array
    */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

Получим в виде коллекций.

```
class User extends Model
{
    /**
    * The attributes that should be cast.
    *
    * @var array
    */
    protected $casts = [
        'images' => 'collection',
    ];
}
```

Вот такой список кастов есть в коробке на момент написания этого гайда:

- array
- AsStringable::class
- boolean
- collection
- date
- datetime
- immutable_date
- immutable_datetime
- decimal:<precision>
- double
- encrypted
- encrypted:array
- encrypted:collection
- encrypted:object
- float
- integer
- object
- real
- string
- timestamp

Но мы также можем писать и свои, выносить их в отдельные классы и указывать в модели в свойстве `$casts`, которое мы обсуждали только что.

Чтобы создать `Cast` можно воспользоваться консолью и artisan:

```
php artisan make:cast PhoneCast
```

Создастся класс `PhoneCast` и размещен он будет в директории `app/Casts`. В нем по умолчанию будет два метода - `get` и `set`, знакомые нам.

Вот так он выглядит:

```php
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class PhoneCast implements CastsAttributes
{
    /**
    * Cast the given value.
    *
    * @param  \Illuminate\Database\Eloquent\Model  $model
    * @param  string  $key
    * @param  mixed  $value
    * @param  array  $attributes
    * @return array
    */
    public function get($model, $key, $value, $attributes)
    {
        return '+' . $value;
    }

    /**
    * Prepare the given value for storage.
    *
    * @param  \Illuminate\Database\Eloquent\Model  $model
    * @param  string  $key
    * @param  array  $value
    * @param  array  $attributes
    * @return string
    */
    public function set($model, $key, $value, $attributes)
    {
        return trim(preg_replace('/^1|\D/', "", $value));
    }
}
```

Если вернетесь выше к Mutators/Accesors, то поймете, что здесь у нас тоже самое, только в отдельном классе. А далее установим его на поле `phone` для модели `User`:

```
use App\Casts\PhoneCast;

class User extends Model
{
    /**
    * The attributes that should be cast.
    *
    * @var array
    */
    protected $casts = [
        'phone' => PhoneCast::class,
    ];
}
```

ГЛАВА 10. SCOPES

https://laravel.com/docs/eloquent#local-scopes

> *«Страус-программист дублирует одинаковые запросы к базе даже если использует ORM. Laravel-разработчик все выносит в scope для удобства.»*

Мы научились пользоваться моделями, умеем работать с Query Builder и строим запросы как по маслу. Давайте рассмотрим пример - мы с вами получаем статьи на главной странице:

```
$posts = Post::query()->where('active', true)->where(
'is_moderated', true)->where('banned', false)->get();
```

Появляется задача сделать запрос на странице статей и где-то еще. Замечаем, что у нас появляется запрос-дубль, который мозолит нам глаза и хотелось бы просто сделать вот так:

```
$posts = Post::query()->active()->get();
```

И на уровне модели это делается очень просто: объявляем метод в котором будет суффикс `Scope` и в нашем случае это:

```
public function activeScope(Builder $query)
{
    $query->where('active', true)->where('is_moderated',
true)->where('banned', false);
}
```

Как видите, мы перенесли все наши условия в один метод и далее, за счет магии Laravel, вызываем его в нашем Query Builder! Обязательно берите на заметку и пользуйтесь!

ГЛАВА 11. VIEWS/BLADE

ГЛАВА 11.1. VIEW

https://laravel.com/docs/views

> *«Страус-программист пишет всю логику в html,*
> *а у Laravel-разработчика все разделено на отдельные файлы*
> *и он руководствуется следующим разделом гайда»*

Помните, мы разбирали с вами роуты и поняли, что за счет них можем выдавать различные ответы пользователю. Мы делаем web-сайт и нам нужно отобразить html страницу. Как раз визуальное представление нашего сайта и обеспечивается view. Вот так выглядит роут, который отображает view:

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

Давайте пока думать, что `view` это html страница, и мы на роуте главной страницы отобразили `home.html`. Сразу назревает вопрос: а где эта html лежит?

Ответ: в Laravel они в директории `/resources/views`. И если мы указали `home`, то файл будет лежать в `/resources/views/home.blade.php` (не пугайтесь что у нас не html, а `blade.php` - уже скоро мы это обсудим). Но прежде еще один пример: если у нас `view` лежит еще в поддиректории, скажем вот тут - `/resources/views/pages/home.blade.php`, тогда роут бы выглядел так:

```
Route::get('/', function(){
    return view('pages.home');
});
```

Мы указываем иерархию каталогов в папке `/resources/views/`, используя в качестве разделителя не `/`, а `.`, и это частая практика в Laravel для вложенных структур, запоминайте.

ГЛАВА 11.2. BLADE

https://laravel.com/docs/blade

> *«Страус-программист пишет огромные view файлы,*
> *а Laravel-разработчик делит все на компоненты*
> *и отдельные view, чтобы не плодить дубли*
> *и сократить структуру приложения»*

Ранее для того, чтобы избежать путаницы, мы решили, что условно `view` это html файлы, а потом увидели что этот html имеет расширение `blade.php`.

Так вот не пугайтесь, блейд это шаблонизатор, который помогает нам генерировать итоговые страницы. В нем присутствует свой синтаксис, который вам еще предстоит изучить. В итоге наш blade-файл будет компилироваться в простой нативный php, который также будет кэшироваться и не повлияет на производительность.

Давайте разберем содержимое blade-файла. В блейдах много удобных особенностей, о которых вы узнаете самостоятельно, когда будете разрабатывать свои крутые проекты. Самое первое, что делает новичок, это выводит данные на страницах. Мы уже знаем о моделях и умеем строить запросы, поэтому мы сможем вывести какую-то информацию.

Но для этого нам нужно не только указать в роуте расположение нашей `view`, нам также туда надо что-то передать для отображения.

Совет - не рекомендуется строить логику во `view`, выносите её в контроллеры (о которых мы скоро поговорим). Поэтому пока что будем писать её на уровне роутов:

```
return view('home', ['posts' => Post::query()->
active()->get()]);
```

Вот так, поместив нужные данные в массив, мы можем передавать во `view` любые данные, в нашем случае это опубликованные статьи.

Пришло время для `blade` шаблона `home`:

```
<html>
<head>
    <title>Cutcode here!</title>
</head>
<body>
    <ul>
        @foreach($posts as $post)
        <li>{{  $post->name }}</li>
        @endforeach
    </ul>
</body>
</html>
```

Согласитесь, здесь нет ничего пугающего. Мы видим html-теги, а также php-код, только конструкции, которые в blade называются директивами, начинаются с символа `@`, но к ним очень просто привыкнуть.

Давайте рассмотрим несколько директив:

```
@if(true)
@endif

@auth
@endauth
```

Я думаю, что с последней директивой вы задумались и не смогли сопоставить её с php-кодом.

Да, друзья, это одна из вспомогательных директив. Например, в `@auth` находится проверка на то, аутентифицирован ли пользова-

тель. Директив очень много и вы познакомитесь с ними в процессе дальнейшего обучения.

Также я думаю, что вам стало интересно, что же это за конструкция `{{ $post->name }}`. И это простая интерполяция. Если вы пришли из мира фронтенда и работали с `vue`, то `blade` покажется вам родным домом, так как разработчики `blade` вдохновлялись `vue` при создании своего шаблонизатора. И `{{ $post->name }}` эквивалентно `echo $post->name`.

Также blade файлы можно разделять на составные части, так называемые партиалы, и делать компоненты. Все это вы узнаете позже, когда изучите этот гайд и окунетесь в мир Laravel с головой.

Друзья, на что еще стоит обратить внимание? Мы с вами выше рассмотрели вывод значений с помощью синтаксиса двойных фигурных скобок - `{{ $post->name }}` и для простоты понимания сопоставили со знакомым `echo $post->name`. Но мы не учли один момент, что в этом случае к тексту применяется функция `strip_tags`, которая очищает строку от всего лишнего. Но если нам потребуется вывести содержимое как оно есть, например для вывода html из поля, содержимое которого мы записали через визуальный редактор в админ панели, то нам нужно будет вывести его с помощью следующего синтаксиса - `{!! $post->text !!}` и он будет работать без `strip_tags`.

Настоятельно рекомендую вам очень аккуратно использовать такую форму вывода, так как в некоторых случаях это может стать местом, уязвимым для xss-атак. Мы еще обсудим это подробнее в главе 16 про безопасность. Вы можете сначала перейти к ней, а потом вернуться сюда и продолжить.

Полезный видео-гайд по использованию blade-компонентов:

https://www.youtube.com/watch?v=vTdH8L45joQ

https://www.youtube.com/watch?v=GgJfHPEFBRs

ГЛАВА 12. КОНТРОЛЛЕР

https://laravel.com/docs/controllers

> *«Страус-программисты пишут логику где попало, можно встретить и во view, и в route и просто где придется, а Laravel-разработчик выносит логику как минимум в контроллеры. Немного позднее вы узнаете еще множество слоев куда выносят логику ниндзя Laravel-разработчики, но всему свое время!»*

Мы оставили контроллеры на десерт не просто так, это наш мост между `route` и `view`. Здесь мы будем принимать запросы и генерировать ответы, в нашем случае ответ в виде `view` с данными. Давайте посмотрим, как устроить этот процесс на основе того, что у нас уже есть. Так сказать проведем рефакторинг.

Для начала создадим контроллер. И делается это с помощью все того же artisan:

```
php artisan make:controller HomeController
```

Это контроллер для нашей главной страницы и появился он в `app/Http/Controllers/HomeController.php`.

Все, что касается взаимодействий по протоколу http, будет хранится в директории `app/Http`. Вот содержимое созданного контроллера:

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

use Illuminate\Http\Request;

class HomeController extends Controller
{
```

```
    //
}
```

Пока он пустой. Нам нужно добавить метод для главной страницы.

Давайте назовем его `index` и перенесем в него логику из роута:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller
{
    public function index()
    {
        return view('home', ['posts' => Post::
query()->active()->get()]);
    }
}
```

Давайте исправим теперь роут чтобы он ссылался на наш контроллер:

```
Route::get('/', [HomeController:class, 'index']);
```

Как видите, вместо замыкания в функцию мы передали массив, где первый элемент это путь до контроллера, а второй - метод в контроллере, который будет обрабатывать запрос. В итоге, при запросе на главную страницу, мы увидим результат скомпилированного `home.blade.php`.

Хотите больше погрузится в тему? Посмотрите видео и узнайте куда следует выносить логику из контроллеров:

https://www.youtube.com/watch?v=0CUIG1JM4IA

ГЛАВА 13. SERVICE CONTAINER

В первой главе мы уже сталкивались с этим термином, когда указывали интерфейс `Kernel` и получали конкретный класс по работе с Http приложением. Давайте немного углубимся в эту тему, ведь она является архитектурной концепцией Laravel и поэтому пройти мимо нее мы просто не можем.

Предлагаю ознакомиться с ней на примере контроллеров, которые мы с вами только что рассматривали. Допустим у нас с вами есть метод в контроллере, который выглядит следующим образом:

```
public function indexPage(Request $request)
{
}
```

или

```
public function indexPage(User $user)
{
}
```

В теле этих методов мы можем получить экземпляр класса по запросам, модель пользователя, или экземпляр любого другого класса, который мы решим там указать. Но как же так? У нас есть аргументы, но мы с вами нигде их не передавали, а просто указали нужный контроллер и его метод в роуте. Еще возможно мы передали параметр `$user` через роут, но мы ведь не сообщали Laravel, что нам нужна модель. Это какая-то магия?

Совсем нет, здесь за работу берется паттерн Инъекция зависимостей (DI), который реализован в Laravel и называется `Service Container`. Через него проходит множество сущностей, таких как контроллеры, события, мидлвары и джобы, или же вы можете создавать свои, но обо всем по порядку.

Все контроллеры проходят через функционал сервис контейнера,

Laravel проходится по методам и конструкторам, и реализует все указанные аргументы, создавая их экземпляры. После этого мы можем с ними спокойно работать. Вы можете поиграться, создав контроллер и делая инъекции, указывая различные классы и получая их экземпляры.

Это хорошая практика, но я уверен, что скоро вы столкнетесь с проблемой. Какой?! Смотрите, допустим у вас есть класс у которого в конструкторе. Присутствуют обязательные аргументы, скажем вот такой:

```
class Car {
    public function __construct(
        protected string $color
    ) {}
}
```

И вот вы продолжаете играться с инъекциями зависимостей и используете класс `Car`:

```
public function indexPage(Car $car)
{
}
```

В итоге получите ошибку. Экземпляр класса не создается, так как требует обязательный параметр `$color`. Все логично, сервис контейнер мощный инструмент, но к сожалению пока без искусственного интеллекта. И понять, что именно нужно указывать во всех параметрах, он не способен. Но это отличный пример для того, чтобы продолжить углубление в эту тему.

Давайте поможем сервис контейнеру Laravel решать подобную проблему. Откроем `app/Providers/AppServiceProvider`. Мы уже рассматривали с вами `RouteServiceProvider`, который отвечал за маршрутизацию, а сейчас смотрим на провайдер, который ответственен за наше приложение в целом. И это отличное место для решения вопросов с сервис контейнером.

```
public function boot(): void
{
    $this->app->instance(Car::class, new Car('white'));
}
```

В методе `boot()` мы обратились к свойству `app` с нашим приложением. И через метод `instance` указали, что если мы делаем инъекцию с классом `Car`, то получать будем экземпляр `Car` с цветом `white`. И теперь в нашем контроллере не будет ошибок.

Но самое интересное еще не это. Мы можем с вами указать не конкретный класс, а интерфейс, но при этом получать экземпляр конкретного класса. Мы уже смотрели с вами на подобные примеры когда рассматривали путь запроса. И видели, что интерфейс `Kernel` иногда был классом `Kernel` по работе с http запросами, а иногда с консольными командами:

```
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);
```

И после мы можем сделать инъекцию в контроллере:

```
public function indexPage(Illuminate\Contracts\Http\Kernel
$kernel)
{
}
```

Но при этом получить экземпляр класса `App\Http\Kernel`.

А также мы можем в процессе выполнения нашего приложения менять реализации с указанием интерфейса. Например, у нас есть интерфейс, который отправляет уведомления в различные мессенджеры. И мы делаем связывание в сервис провайдере, которое выглядит следующим образом:

```
public function boot(): void
{
    $this->app->bind(MessengersInterface::class, Telegram::
class);
}

public function indexPage(MessengersInterface $messenger)
{
}
```

Делаем инъекцию и получаем класс, который отправляет уведомления в Telegram. Со временем решили поменять Telegram на Slack, и нам с вами не придется бегать по всем контроллерам и прочим сущностям и менять инъекцию с Telegram на Slack. Мы просто откроем сервис провайдер и поменяем:

```
$this->app->bind(MessengersInterface::class, Slack::
class);
```

Теперь наше приложение переключится, но самое интересное заключается в том, что то же самое мы можем делать и в процессе исполнения, а не только в сервис провайдере, и на определенное условие изменить `bind`. И далее при последующих инъекциях получать указанный экземпляр.

Получить экземпляр класса в рамках сервис контейнера можно не только через инъекцию, а также через хелпер `app()`:

```
public function indexPage()
{
    $messenger = app(MessengersInterface::class);
}
```

Самые внимательные из вас я думаю заметили, что мы с вами работаем для связывания абстракции и конкретного класса через метод `bind`. А вот в примере на пути запроса был `singleton`. Но на самом деле внутри метода `singleton` тоже метод `bind`, но с указанием что мы будем сохранять экземпляр в памяти, и если экземпляр уже создавался ранее, то по новой он создан не будет, в отличии от простого `bind` (за это отвечает реализация паттерна `singleton`).

Еще больше информации о Service Container, а также о паттерне dependency injection (инъекция зависимостей) - как это все работает с разбором на примерах смотрите в видеоролике:

 https://www.youtube.com/watch?v=-38OQNb8ZWg

ОРАНЖЕВЫЙ ПОЯС

ГЛАВА 14. ДВА БРАТА REQUEST И RESPONSE

https://laravel.com/docs/requests

https://laravel.com/docs/responses

Помните главу про путь запроса `(Request)` ? В текущей главе мы разберем объект запроса, а также `Response` - брата `Request` .

Нам постоянно приходится работать с запросами и ответами на них. Для нашего с вами удобства в Laravel есть объекты для взаимодействия с этими явлениями, и сейчас мы их рассмотрим.

Мы уже с вами работали с роутами, дополняли контроллер и знаем, что во главе этого пути стоит запрос.

Мы можем в контроллере в нашем методе `index` сделать инъекцию с объектом `Request` , в котором будет вся информация о запросе, и взаимодействовать с ним, либо воспользоваться хелпером `request()` и получить на выходе то же самое. Глянем на примере:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller
{
    public function index(Request $request)
    {
        // все параметры реквеста
        dump($request->all());

        // получим метод запроса GET или POST PUT DELETE и
остальные
        dump($request->method());

        // Читаем куку
        dump($request->cookie('name'));

        // Берем файл
        dump($request->file('file'));

        // Заголовок
        dump($request->header('name'));

        return view('home', ['posts' => Post::
query()->active()->get()]);
    }
}
```

Все, что касается запроса, мы можем получить из этого объекта с помощью удобных методов.

А теперь давайте займемся его братом - ответом (`Response`).
С ним мы уже также работали, когда отдавали view и это был 200 ответ с html контентом.

Ответ также мог быть редиректом:

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

Или json:

```
return response()->json(['data' => '']);
```

Как видите, в Laravel разнообразие хелперов по генерации ответов на любой случай жизни!

ГЛАВА 15. ВАЛИДАЦИЯ

 https://laravel.com/docs/validation#main-content

Валидация представляет собой процесс проверки данных различных типов на соответствие критериям корректности. Она используется в работе, чтобы не пропускать всякую нечисть, так как запросы могут быть плохими или не валидными.

Представьте ситуацию: у нас есть `view`, это контактная форма, в которую пользователь вводит данные: email, телефон и так далее. Нам нужно убедиться, что все поля заполнены, что email и телефон валидные (ведь пользователь может набезобразничать и ввести некорректные данные). Также в наших силах сразу проверить, что указанных пользователем данных нет в базе и ранее он к нам не обращался. Как же выполнить все проверки с введенными данными, а также обезопасить себя от не валидного и, возможно, вредоносного контента?

Делается это несколькими путями, но в основе будет класс `Validator`.

Первый способ - валидация объекта с помощью реквеста (запроса). Допустим, при сохранении статьи вам нужно выполнить проверку, существует ли уже статья с таким названием. Давайте сделаем запрос, проверим параметр `title` - он должен обязательно присутствовать, быть уникальным (то есть другого такого же тайтла в таблице `posts` быть не должно) и по количеству символов не превышать 255 символов:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller
{
    public function index(Request $request)
    {
        $validatedData = $request->validate([
   'title' => ['required', 'unique:posts', 'max:255'],
   'body' => ['required'],
        ]);
    }
}
```

Как видите, мы проверяем параметры пришедшего запроса на соответствие определенным правилам. Есть очень много уже готовых правил и вы обязательно с каждым познакомитесь в процессе освоения документации и практики работы с Laravel.

Если все хорошо, то мы пройдем по контроллеру дальше, а если нет, то Laravel сделает редирект на предыдущую страницу с формой, а в сессию сохранит ошибки валидации. А мы уже знаем про `view` и `blade`, а теперь еще можем обработать там ошибки с помощью директивы `@error`:

```
<input type="text" name="title" />
@error('title')
{{ $message }}
@enderror
```

Тем самым мы проверили, есть ли ошибки для поля `title`, и если есть, то отобразили сообщение. Точно в такой же манере мы можем добавить классов инпуту и сделать его красным, вывести уведомление и показать текстовую ошибку, как в примере выше.

И эта ошибка будет соответствовать набору правил, которые не были пройдены в процессе валидации.

Чтобы не создавать кашу кода в одном месте (в нашем случае в контроллере, хоть контроллер и место для нашей логики, он все равно должен быть тонким и не перегружен), всю логику по валидации рекомендуется располагать в отдельных классах. Валидация не исключение и в Laravel есть специальные `FormRequest` классы для этого.

Давайте создадим такой и снова сделаем рефакторинг

```
php artisan make:request ContactFormRequest
```

По этой команде будет создан файл `ContactFormRequest.php` и располагается он в директории `app/Http/Requests/` (как и положено всему по http).

Содержимое `ContactFormRequest.php`:

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactFormRequest extends FormRequest
{
    /**
    * Determine if the user is authorized to make this re-
quest.
    *
    * @return bool
    */
    public function authorize()
    {
        return true;
    }

    /**
    * Get the validation rules that apply to the request.
    *
    * @return array<string, mixed>
    */
    public function rules()
    {
      return [
        'title' => ['required', 'unique:posts', 'max:255'],
        'body' => ['required'],
      ];
    }
}
```

Как видите, мы просто перенесли правила валидации из контроллера в этот класс и имеем еще больше контроля в рамках методов этого класса, которые мы можем переопределять.

Хочу также обратить ваше внимание на метод authorize, который мы установили в true. По умолчанию, при генерации FormRequest он выставлен в false, а этот метод ограничивает доступ к выполнению http запроса. И если мы укажем false, то пользователь даже не дойдёт до валидации, а ответ сервера сразу будет 403 - доступ запрещён. Вы можете добавлять логику авторизации и, например, запретить запрос от неавторизованного пользователя.

```
public function authorize(): bool
{
    return !auth()->guest();
}
```

Теперь давайте изменим контроллер:

```
<?php

namespace App\Http\Controllers;

use App\Http\Request\ContactForm;

class HomeController extends Controller
{
    public function index(ContactForm $request)
    {

    }
}
```

Смотрите - мы просто сделали инъекцию в метод с нашим реквест классом. Под капотом Laravel сам вызовет метод `validate` и в случае ошибок сделает редирект. Все это будет сделано за нас, мы только добавили его в качестве аргумента.Как красив и прост наш код, ну очень лаконично!

ГЛАВА 16. БЕЗОПАСНОСТЬ

https://laravel.com/docs/csrf

> *«Страус-программисту плевать на безопасность, если что - голову в песок и плевать на проблемы, авось пронесёт! Laravel-разработчик думает о безопасности и у него есть для этого все инструменты.»*

Да, друзья, мы используем Laravel и его создатели на каждом шагу пытаются подстраховать нас в вопросах безопасности, но мы и сами никогда не должны о ней забывать. Не стоит полагаться на кого-то, и следует всегда соблюдать правила безопасности в своих проектах.

Давайте для начала обсудим основные проблемы с которыми мы можем столкнуться, а также научимся их избегать.

ГЛАВА 16.1. SQL ИНЪЕКЦИЯ

Это угроза безопасности происходит, когда злоумышленник пытается ввести sql код в какой-либо `input` на вашем сайте. Например, в контактной форме, или в профиле пользователя в текстовое поле «обо мне», в общем - в любом месте, где мы даем пользователю редактировать информацию в нашей базе данных. Рассмотрим пример

```
$user['about'] = $_POST['about'];
$query = "UPDATE users SET about  = '".$user['about']."'";
```

Если ваш код выглядит как в примере выше, а как правило страус-программисты пишут именно так, то пользователю не составит труда в поле обо мне добавить текст вида:

```
test' AND 'is_admin' = "1
```

и итоговый sql будет иметь вид:

```
// UPDATE users SET about  = "test" AND is_admin = "1"
```

Таким образом пользователь сделал себя админом и это наименьшая из бед, которую он мог бы сделать, ведь можно продолжить запрос и удалять таблицы или всю базу. Вот тогда будет совсем плохо, согласитесь.

Но выдыхаем и радуемся, что мы теперь ниндзя Laravel-разработчики. И благодаря Eloquent мы застрахованы от SQL инъекций, так как Laravel будет биндить (связывать) значения и удалять лишнее и потенциально опасное, включая инъекции. А благодаря свойству `fillable` в моделях мы также будем контролировать, чтобы пользователи не могли менять поля, которые не следует. Как например `is_admin` из примера выше.

ГЛАВА 16.2. CROSS SITE SCRIPTING

Еще одна проблема которая похожа на первую. Но также будет отсылкой к главе 11 с blade шаблонизатором, где мы уже начинали обсуждать эту проблему.

Допустим, все еще профиль пользователя и поле «Обо мне» (`about`) , которое мы даем заполнять. И юзер может нам навредить и без sql инъекций, просто добавив js скрипт, который может быть немного хулиганским, как ниже:

```
<script>alert('hello, I just hacked this page');</script>
```

А может и делать редиректы на вредоносные источники и так далее.

Но благодаря `blade` и синтаксису интерполяции с экранированием тегов, мы застрахованы от таких сценариев:

```
{{ $user->about }}
```

А вот такой синтаксис, как мы писали выше, не будет экранировать теги:

```
{!! $user->about !!}
```

В таком случае можно будут запускаться все скрипты, которые добавит пользователь, поэтому всегда помните об этом и давайте выводить значение поля только при полном контроле наполнения.

Теперь мы знаем как предотвратить любой межсайтовый скриптинг (так называемая атака XSS).

ГЛАВА 16.3. CROSS-SITE REQUEST FORGERY (CSRF)

https://laravel.com/docs/csrf

Подделывание межсайтовых запросов.

Давайте начнем с возможного сценария прямо из документации Laravel, на мой взгляд отличный пример. У нас есть `route` - `/user/email` который меняет e-mail аутентифицированного пользователя и форма на вашем сайте крайне простая. Выглядит вот так:

```
<form action="https://your-application.com/user/email"
method="POST">
   <input type="email" value="malicious-email@example.com">
</form>
```

Ок, все работает, пользователь сможет заменить данные когда захочет. Он аутентифицирован на вашем сайте, но злодей-хакер заманивает его на свой сайт, где он использует вашу верстку и сэмулировал эту страницу сайта один в один! И ничего не подозревающий пользователь заполняет данные, но злоумышленник их подменяет на свои и отправляет на ваш сайт, где пользователь аутентифицирован и меняет email пользователя на email хакера! И теперь хакер сменил email пользователя на свой, делает восстановление пароля и получает доступ к вашему аккаунту. Ужас, да? И страус-программист легко бы такое допустил, не догадываясь, что в Laravel есть простой механизм защиты от csrf атак.

Чтобы обезопасить себя от подобных атак, нужно всегда помнить, что для сохранения данных нельзя использовать метод `GET`, а только `POST|PUT|DELETE`. Эти методы в Laravel защищены мидлваром `VerifyCsrfToken` который располагается в `app/Http/Middleware` и в `app/Http/Kernel.php`, и установлен по умолчанию в группу `web` (обязательно убедитесь, что он присутствует на вашем web-сайте и защищает вас от csrf атак).

Далее все формы на сайте с методами отправки `POST|PUT|DELETE`

должны иметь поле с Laravel csrf токеном с помощью директивы `@csrf` или с помощью хелпера `csrf_token()`:

```
<form method="POST" action="/profile">
    <!-- Выведет hidden input с токеном как и пример ниже
... -->
    @csrf

    <!-- Тоже самое с помощью токена ... -->
    <input type="hidden" name="_token" value="{{
csrf_token() }}" />
</form>
```

После этого все ваши формы будут проверяться на бекенде с сессией, в которой также присутствует токен. Запросы без этого токена будут блокироваться мидлваром. Похлопаем себе и Laravel, мы в безопасности, как и наши клиенты.

Если Вы хотите узнать больше о безопасности Ваших проектов на Laravel, то рекомендую посмотреть ролик о Middlewares безопасности на моём канале:

https://www.youtube.com/watch?v=A--rHfjhmUc

ГЛАВА 17. SESSION

https://laravel.com/docs/session

> *«Страус-программист каждый раз обращается к сессиям из глобальной переменной, тем самым лишая код гибкости и привязываясь к единой реализации. Laravel-разработчик использует специальный класс по работе с сессиями и в любой момент может изменить хранилище сессий.»*

Сессии помогают нам передавать некоторые данные о пользователе между запросами во время посещения сайта, чтобы сохранять необходимые нам состояния. В Laravel есть удобный механизм для работы с сессиями.

Для начала необходимо в конфиге `app/session.php` указать, как именно будут храниться сессии. Нам доступно несколько вариантов:

- file - по умолчанию, стандартный вариант, где сессии хранятся в виде файлов в storage/framework/sessions.
- cookie - сессии хранятся в безопасных закодированных cookie.
- database - сессии будут храниться в базе данных
- memcached / redis - сессии будут хранится в памяти, крайне быстрый способ получения и сохранения данных
- dynamodb - в AWS DynamoDB.
- array - по большему счету нужно для тестов где сессии представлены в виде php массива и не сохраняются.

Вы в любой момент можете изменить драйвер, а с помощью хелпера `session()` удобно записывать, получать и строить логические цепочки с сессиями.

Давайте рассмотрим несколько примеров:

```
session()→put('name', 'value');
// записали значение в сессию с ключом name
session()→get('name');
// получили значение value
if(session()→has('name')) {}
// проверили есть ли в сессиях такие данные
```

И множество других удобных методов, с которыми вы в процессе познакомитесь. Продублирую ссылку на документацию, ведь это основа обучения

 https://laravel.com/docs/session

ГЛАВА 18. АУТЕНТИФИКАЦИЯ

> *«Аутентификация - большая боль для страус-программиста, на реализацию которой он может убить несколько недель. А мы (Laravel-разработчики) напишем аутентификацию за пару минут (ну максимум за 5).»*

Друзья, в рамках аутентификации никаких велосипедов нам не потребуется. В Laravel уже все подготовлено для того чтобы реализовать вход, регистрацию, восстановление пароля и прочее. И все это настолько просто, что реализация аутентификации в проектах не станет для вас рутиной.

А чтобы это не превращалось в однообразную работу по копированию кода, разработчики Laravel также подготовили для нас готовые решения, а именно:

1. Простой `breeze`

https://laravel.com/docs/starter-kits#laravel-breeze

2. Более расширенный `JetStream`

https://jetstream.laravel.com/2.x/introduction.html

Вы можете использовать их и не изобретать даже тот простой велосипед, который уже есть в Laravel. Или подсмотреть реализации в github репозиториях этих проектов и сделать под себя.

Но давайте взглянем как бы выглядел процесс входа на нашем проекте, если бы мы делали аутентификацию сами с использованием базовых инструментов Laravel.

Допустим у нас есть форма входа с двумя полями email и пароль. На бекенде проверка будет выглядеть вот так:

```
if(auth()->attempt([
    'email' => request('email'),
    'password' => request('password')
])) {
}
```

Не поверите, но это все! Метод `attempt` сам проверит, есть ли такой юзер, сделает хеш пароля и сверит его с тем, что есть в базе. И если все совпадает, то аутентифицирует юзера и создаст ему сессию. А регистрация это просто создание пользователя с помощью Eloquent:

```
User::query()->create([
    'email' => request('email'),
    'password' => Hash::make('password')
]);
```

Тут также вспоминаем главу 15 с валидацией, и перед сохранением обязательно проверяем данные. Единственное отличие заключается в том, что мы не просто сохраняем пароль в том виде, в каком вводит юзер, а создаем его хеш, чтобы никто не имел прямого доступа к паролю. А с помощью ключа `APP_KEY` в `.env` Laravel будет проверять пароли и аутентифицировать пользователя.

Еще мастхэв методы:

- Выход из сайта:

```
auth()->logout();
```

- Аутентификация самостоятельно имея модель `User`:

```
auth()->login($user);
```

- Аутентификация с помощью `id` пользователя:

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

Ну и немного инструментов из других глав в связке с аутентификацией.

Мидлвар `auth` говорит о том, что доступ к роуту будут иметь только аутентифицированные пользователи:

```
Route::get('/profile', ProfileController::class)->
middleware('auth');
```

А `guest` - только гости:

```
Route::get('/profile', ProfileController::class)->
middleware('guest');
```

Пример из `blade`:

```
@auth
Я {{ auth()->user()->name }} и мой id - {{ auth()->id() }}
@endauth

@guest
А здесь будет кнопка “Войти”
@endguest
```

Это примеры с условными директивами. В первом примере если пользователь авторизован, то выводим его имя и `id` (заодно глянули как это делать через `helper`).

Для гостей вывели кнопку «Войти». Думаю в первый пример еще можно добавить кнопку «Выйти». Что скажете на это?

Как реализовать аутентификацию на реальном проекте можно посмотреть в этом видео:

https://www.youtube.com/watch?v=DGaXW6DLV6M

УРОВЕНЬ 4.

КРАСНЫЙ ПОЯС

ГЛАВА 19. КОНСОЛЬНЫЕ КОМАНДЫ

https://laravel.com/docs/artisan

> *«Страус-программист каждый раз все делает вручную, а ниндзя Laravel-разработчик использует консольные artisan команды. Хороший парень!»*

Мы уже работали с консольными командами на протяжении этого гайда. Когда создавали контроллер:

```
php artisan make:controller HomeController
```

Или когда делали каст и form request:

```
php artisan make:cast PhoneCast
php artisan make:request ContactFormRequest
```

Или же просто запускали виртуальный сервер:

```
php artisan serve
```

Или накатывали миграции:

```
php artisan migrate
```

Ну вообщем вы поняли! Мы упрощали себе жизнь и экономили время.

Но мы также можем писать и свои консольные команды. Чтобы создать такую команду, нужно также воспользоваться artisan:

```
php artisan make:command CreateTestUse
```

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class CreateTestUser extends Command
{
    /**
    * The name and signature of the console command.
    *
    * @var string
    */
    protected $signature = 'command:name';

    /**
    * The console command description.
    *
    * @var string
    */
    protected $description = 'Command description';

    /**
    * Execute the console command.
    *
    * @return int
    */
    public function handle()
    {
        return Command::SUCCESS;
    }
}
```

Вот так она выглядит. Обратите внимание на свойство `$signature`. Нам оно потребуется чтобы вызвать эту команду из консоли. И если исходить из варианта по умолчанию, то нам нужно написать следующее:

```
php artisan command:name
```

Нажав `enter`, мы выполним содержимое метода `handle`. Согласитесь, странное название команды. И сама команда пока пустая. Давайте исправим это, чтобы еще глубже понять тему:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class CreateTestUser extends Command
{
    /**
    * The name and signature of the console command.
    *
    * @var string
    */
    protected $signature = 'create:user';

    /**
    * The console command description.
    *
    * @var string
    */
    protected $description = 'Create test user';

    /**
    * Execute the console command.
    *
    * @return int
    */
    public function handle()
    {
        User::create([
            'email' => 'test@example.com'
        ]);

        return Command::SUCCESS;
    }
}
```

Теперь мы можем развернуть проект и выполнить:

```
php artisan create:user
```

И сразу создать нам тестового юзера для дальнейшего взаимодействия с сайтом. Работать с консольными командами несложно, а будет еще проще, когда мы изучим тему фабрик и сидов, которым посвящена следующая глава.

ГЛАВА 20. СИД И НЭНСИ? ПОЧТИ! СИДЫ И ФАБРИКИ

ГЛАВА 20.1. СИДЫ

 https://laravel.com/docs/seeding

> *«Страус-программист делает sql дампы с дополнительной и тестовой информацией, каждый раз выдумывает данные для тестов и создает их с нуля на тестовом окружении вручную. Звучит это ужасно, а на практике еще хуже. Мы же не динозавры, и не страус-программисты, знаем как работать с тестовыми базами данных и добавляем их в проект одной командой.»*

Чуть выше мы с вами создали тестовые данные с юзером с помощью консольной команды. Но есть и более изящный способ и отдельная для этого ответственность. Это сиды.

Мы с вами в отдельном классе (или классах) можем создать все необходимые тестовые данные для нашего проекта и локальной разработки, либо создать важные для проекта данные по-умолчанию (такие как статусы заказа, товаров и так далее). А создавать и добавлять в базу эти данные мы будем также через консольные artisan команды.

Друзья, с одной стороны сиды лучше бы перенести в главу с миграциями, но я считаю, что сиды нужны для тестовых данных. А фиксированные данные лучше добавлять через миграции (в официальной документации они находятся в разделе базы данных, а раньше были в разделе тесты). И если сиды нужны для тестов и тестового наполнения сайта, то без фабрик сиды жить не могут. Скажем так: сиды без фабрик - деньги на ветер! Поэтому я и решил объединить их в одну главу. Кстати, фабрики раньше были тоже в разделе тесты и там им и место на мой взгляд, но сейчас перекочевали в раздел eloquent model.

Давайте рассмотрим один из простых примеров, когда нам это мо-

жет потребоваться: мы разрабатываем проект, и чтобы понять, что он работает правильно, что верстка нигде не едет - мы наполняем его тестовым контентом. С нами в команде работают еще разработчики, и когда они развернут проект у себя, у них также по-умолчанию будут и тестовые данные, им не придется задумываться об этом: развернул и можно начинать работать. Это удобно.

Давайте разберемся, где и как эти сиды создавать, и в итоге запустим их для добавления информации в нашу базу данных.

Отправная точка это класс `DatabaseSeeder`, который располагается в `database/seeders/DatabaseSeeder.php`. Вот так он выглядит по умолчанию:

```
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\
WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
    * Seed the application's database.
    *
    * @return void
    */
    public function run()
    {
        // \App\Models\User::factory(10)->create();
    }
}
```

Он пустой, но Laravel нам дает закомментированный пример кода в котором с помощью фабрик (которые скоро обсудим) создадим 10 тестовых юзеров.

Давайте немного подкорректируем класс для лучшего понимания:

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
     public function run()
     {
         OrderStatus::query()->create(['name' => 'new']);
     }
}
```

В данном случае мы создали статус заказа. Теперь при разворачивании проекта и выполнении команды:

```
php artisan migrate -seed
```

будет также запускаться метод run этого класса и добавляться статус заказа - здесь может быть очень много данных, которые требуются проекту для разработки.

Еще одна команда, чтобы запустить сиды без миграций:

```
php artisan db:seed
```

В некоторых проектах сидов может быть огромное количество, и класс, который мы рассматривали выше, разрастется кодом. Давайте создадим отдельный класс сидов под определенную ответственность, пусть это будут статусы заказов. Создадим через артисан команду:

```
php artisan make:seeder OrderStatusSeeder
```

Выглядит точно так же, как и класс, что мы рассматривали выше и

там же располагается, но давайте его сразу подправим под наши нужды:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class OrderStatusSeeder extends Seeder
{
    /**
    * Run the database seeds.
    *
    * @return void
    */
    public function run()
    {
        DB::table('order_statuses')->insert(['id' => 1,
'name' => 'Новый']);

        DB::table('order_statuses')->insert(['id' => 2,
'name' => 'В обработке']);

        DB::table('order_statuses')->insert(['id' => 3,
'name' => 'Подтвержден']);

        DB::table('order_statuses')->insert(['id' => 4,
'name' => 'Оплачен']);
   }
}
```

А далее в основной класс `DatabaseSeeder` мы добавим вызов этого класса. И, как видите, здесь мы уже работали не от модели, а от `DB` фасада, нам никто не запрещает так делать:

```
public function run()
{
    $this->call([
        OrderStatusSeeder::class,
    ]);
}
```

Разделяя сиды на отдельные классы, мы можем выбирать какие именно запускать, делается это вот так:

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

ну или:

```
php artisan migrate --seed --seeder=OrderStatusSeeder
```

Отличный инструмент, упрощающий нам разработку, обязателен к использованию.

Мы выше рассмотрели, что сиды по большей части привязаны к фиксированным данным, необходимым для жизни проекта. А вот если говорить о тестовых контент данных, то тема фабрик упрощает нам процесс троекратно.

ГЛАВА 20.2. ФАБРИКИ

https://laravel.com/docs/eloquent-factories

Теперь давайте поговорим о фабриках, а потом соединим их с полученными знаниями о сидах.

Вы знаете, что такое библиотека `faker`? Если нет, то не переживайте, тут нет ничего сложного. Библиотека `faker` - инструмент, облегчающий нам жизнь, который генерирует случайные данные. Например, какой-либо фейковый текст, имена пользователей, емейлы, изображения и огромное количество данных разных типов и принадлежностей.

Фабрики в Laravel используют библиотеку `faker`. Давайте немного вернемся назад и вспомним модели. Как вы уже узнали, модель это объектное представление для таблиц в базе данных, а таблицы содержат данные. С помощью моделей мы взаимодействуем с данными в таблицах. Фабрики привязываются к моделям, и с их помощью мы можем быстро генерировать фейковые данные для наших моделей (таблиц), а помогает им в генерации библиотека `faker`.

Давайте рассмотрим, как это выглядит. В Laravel есть фабрики по умолчанию для пользователей. Располагаются они в директории `database/factories`, и здесь необходимо вспомнить конвенцию наименований - фабрика должна иметь такое же название как и модель, только с припиской `Factory`. И тогда Laravel найдет ее без труда для нашей модели. Но модель должна содержать трейт `HasFactory` (но он и так есть по умолчанию).

Давайте взглянем на содержимое фабрики для пользователя:

```php
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
```

```
use Illuminate\Support\Str;

/**
* @extends \Illuminate\Database\Eloquent\Factories\Facto-
ry<\App\Models\User>
*/
class UserFactory extends Factory
{
    /**
    * Define the model's default state.
    *
    * @return array<string, mixed>
    */
    public function definition()
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => '$2y$10$92IXUNpkjO0rOQ5byMi.
Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }
    /**
    * Indicate that the model's email address should be
unverified.
    *
    * @return static
    */
    public function unverified()
    {
        return $this->state(fn (array $attributes) => [
            'email_verified_at' => null,
        ]);
   }
}
```

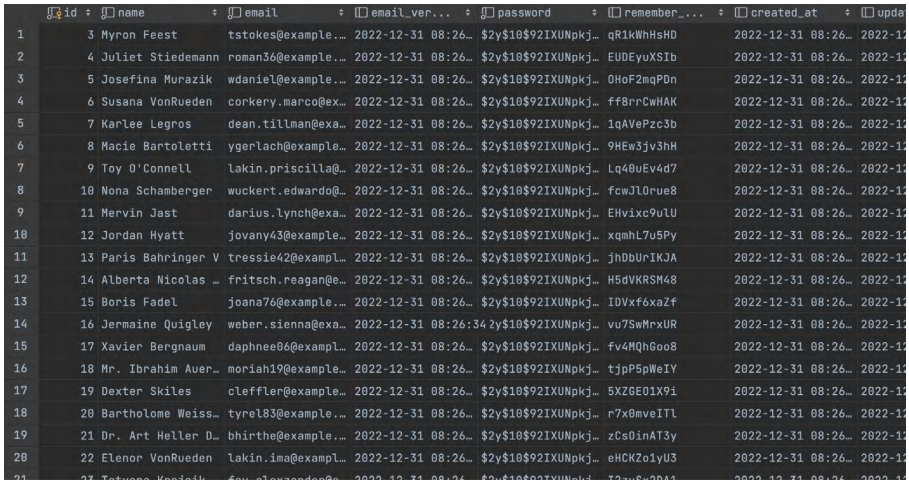
Хелпер `fake()` как раз взаимодействует с библиотекой `faker`. С его помощью мы можем указать, какие именно данные будем генерировать в рамках полей таблицы или атрибутов модели: где-то это имя пользователя, а где-то уникальный email. Я думаю, вы поняли о чем речь.

А теперь, возвращаясь к нашим сидам и классу `DatabaseSeeder`, давайте сгенерируем 100 пользователей. В дальнейшем мы выведем список пользователей на сайте, чтобы убедиться, что все запросы оптимизированы, верстка не разваливается и было с чем работать (как пример):

```
<?php

namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\
WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
    * Seed the application's database.
    *
    * @return void
    */
    public function run()
    {
        \App\Models\User::factory(100)->create();
    }
}
```

Запускаем сиды:

```
php artisan db:seed
```



	id	name	email	email_ver...	password	remember_...	created_at	upda
1	3	Myron Feest	tstokes@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	qR1kWhHsHD	2022-12-31 08:26…	2022-12
2	4	Juliet Stiedemann	roman36@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	EUDEyuXSIb	2022-12-31 08:26…	2022-12
3	5	Josefina Murazik	wdaniel@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	OHoF2mqPDn	2022-12-31 08:26…	2022-12
4	6	Susana VonRueden	corkery.marco@ex…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	ff8rrCwHAK	2022-12-31 08:26…	2022-12
5	7	Karlee Legros	dean.tillman@exa…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	1qAVePzc3b	2022-12-31 08:26…	2022-12
6	8	Macie Bartoletti	ygerlach@example…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	9HEw3jv3hH	2022-12-31 08:26…	2022-12
7	9	Toy O'Connell	lakin.priscilla@…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	Lq40uEv4d7	2022-12-31 08:26…	2022-12
8	10	Nona Schamberger	wuckert.edwardo@…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	fcwJlOrue8	2022-12-31 08:26…	2022-12
9	11	Mervin Jast	darius.lynch@exa…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	EHvixc9ulU	2022-12-31 08:26…	2022-12
10	12	Jordan Hyatt	jovany43@example…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	xqmhL7u5Py	2022-12-31 08:26…	2022-12
11	13	Paris Bahringer V	tressie42@exampl…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	jhDbUrIKJA	2022-12-31 08:26…	2022-12
12	14	Alberta Nicolas …	fritsch.reagan@e…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	H5dVKRSM48	2022-12-31 08:26…	2022-12
13	15	Boris Fadel	joana76@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	IDVxf6xaZf	2022-12-31 08:26…	2022-12
14	16	Jermaine Quigley	weber.sienna@exa…	2022-12-31 08:26:34	2y$10$92IXUNpkj…	vu7SwMrxUR	2022-12-31 08:26…	2022-12
15	17	Xavier Bergnaum	daphnee06@exampl…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	fv4MQhGoo8	2022-12-31 08:26…	2022-12
16	18	Mr. Ibrahim Auer…	moriah19@example…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	tjpP5pWeIY	2022-12-31 08:26…	2022-12
17	19	Dexter Skiles	cleffler@example…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	5XZGEO1X9i	2022-12-31 08:26…	2022-12
18	20	Bartholome Weiss…	tyrel83@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	r7x0mveITl	2022-12-31 08:26…	2022-12
19	21	Dr. Art Heller D…	bhirthe@example.…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	zCs0inAT3y	2022-12-31 08:26…	2022-12
20	22	Elenor VonRueden	lakin.ima@exampl…	2022-12-31 08:26…	$2y$10$92IXUNpkj…	eHCKZo1yU3	2022-12-31 08:26…	2022-12

Теперь все 100 юзеров находятся в нашей базе и, как видите, мы не участвовали в наполнении таблицы. Но фейкер и фабрики поработали на славу, придумывали нам уникальные данные о пользователях, с которыми можно работать и использовать в тестировании (скоро и о тестах мы поговорим).

Фабрики это большой и удобный функционал. В процессе взаимодействия с ними вы разберетесь, как создавать стейты, работать с отношениями и многое другое. Не игнорируйте эти полезные инструменты, и скоро вы не сможете представить разработку без них.

Фабрики и сиды подробно разобраны в небольшом видеогайде на CutCode:

https://www.youtube.com/watch?v=tkyBM-LIc5U

https://www.youtube.com/watch?v=FfBthRRmkQM

ГЛАВА 21. ТЕСТЫ

https://laravel.com/docs/testing

> *«Страус-программист ничего не проверяет и отправляет в продакшен на авось. Если что-то ломается, то часами разбирается в причинах проблем. Но Laravel-разработчик себе такого позволить не может, у него кроме работы есть личная жизнь, поэтому он пишет тесты и имеет контроль над тем как, ведет себя приложение.»*

Друзья, тесты нужны нам, чтобы контролировать поведение нашего проекта, и не бояться вносить изменения. Ведь так часто бывает, что изменяя одну часть нашего проекта, мы забываем о другой, которая зависит от первой. В итоге в проекте появляется ошибка.

Мы видим ошибку, но не понимаем, что случилось, и чтобы определить причину, нам, как правило, потребуется большое количество времени. Имея тесты, мы просто запускаем их в консоли, и, если наш проект достаточно ими покрыт, то мы будем уверены, что все хорошо, и внесенные изменения ничего не поломали. Экономим свое время и нервы. Еще один инструмент, облегчающий нам жизнь и ускоряющий разработку. Хотя в процессе и придется тратить время на написание тестов, но с опытом вы будете делать это с закрытыми глазами.

Однако вы изначально должны выработать в себе привычку, что тесты и рефакторинг это неотъемлемая часть разработки, а не какой то необязательный факультативчик.

Во время тестов мы можем пройтись по всему нашему проекту, нажать каждую кнопку, выполнить все действия, проверить все классы и данные, другими словами полный контроль логики исполнения.

Но, к сожалению, страус-программисты игнорируют тесты, и нам приходится крайне тяжело работать после них в больших проектах где тестов нет. И после каждого изменения кода тратить часы, а то

и дни на проверку функционала сайта.

Давайте напишем простой тест, чтобы понять особенности работы с ними. А со временем вы будете развиваться, эволюционировать, и ваш навык тестирования также будет расти.

Чтобы создать класс с тестами необходимо выполнить команду:

```
php artisan make:test ArticlesTest
```

После выполнения команды класс появится в директории `tests\Feature` и с помощью него мы будем тестировать функционал по статьям.

```
<?php
namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class ArticlesTest extends TestCase
{
    /**
    * A basic feature test example.
    *
    * @return void
    */

    public function testExample()
    {
        $response = $this->get('/');
        $response->assertStatus(200);
    }
}
```

По умолчанию уже даже есть метод с `GET` запросом на главную страницу. Давайте его выполним с помощью команды:

```
$ php artisan test --filter ArticlestTest
```

Опцией – `filter` мы указываем, что нужно выполнить только этот класс. И если опцию убрать, то выполнятся все тестовые классы.

После выполнения мы увидим сообщение в консоли:

```
OK (1 test, 1 assertion)
```

Это говорит нам о том, что все тесты пройдены. Тесты должны быть зелеными и это хорошо. Этим простым тестом мы протестировали, что главная страница доступна и отдает код 200. Простейший тест, но иногда даже простые тесты с проверкой что все страницы работают могут спасти вам кучу времени.

Давайте немного изменим наш тестовый класс, ведь он у нас не про главную страницу, удалим `textExample` и создадим `testArticlesPage`:

```
public function testArticlesPage()
    {
        $this->get('/articles)
            ->assertSee('All articles');
    }
```

В данном тесте мы проверили, что на странице со статьями присутствует заголовок `'All articles'`. Таким образом, мы не только проверяем, что страница работает, а также сверяем ее содержимое с эталоном. Представьте, что теперь мы можем использовать для тестов тестовую базу данных, которую будем каждый раз чистить. И с помощью фабрик в тестах генерировать тестовые статьи и проверять сколько их выводится, в каком они порядке, фильтруются ли они и так далее.

Тесты можете воспринимать как игру, и если они зеленые, то вы выиграли, а если красные то нужно еще потрудиться. Зато вы всегда знаете о проблемах вашего проекта, спокойно улучшаете его, работая в одиночку или в команде. И даже если новичок в вашей команде где-то нахулиганит, с помощью тестов вы всегда узнаете, где и что именно сломалось.

Этой главой мы завершаем наш гайд, но тема тестов очень глубокая и вам предстоит еще погрузиться в нее с головой. Но помните, чем лучше вы пишите тесты и чем больше кода покрываете, тем вы становитесь более сильны как разработчик. Вы и сами в этом убедитесь. Да прибудет с вами сила тестов, мышление разработчика и здравый смысл.

Приглашаю разобраться в теме тестов более подробно. Смотрите мини-курс по тестам на моём канале:

- https://www.youtube.com/watch?v=rEZolULXhhw
- https://www.youtube.com/watch?v=90_hBNcT9HA
- https://www.youtube.com/watch?v=5DFfCzJDiYs

АУТРО

> *«Страус-программист не читает гайды и думает, что он и так знает все, но Laravel-разработчик постоянно развивается и учится. Он получает удовольствие от этого процесса.»*

Друзья, несмотря на то, что вы изучили этот небольшой гайд, я не могу сказать что Вы получили заветный черный пояс по Laravel. Но не расстраивайтесь, я объясню почему. У нас, у разработчиков, есть особенность: наше обучение бесконечно. Вот и я пишу для вас этот гайд и все еще не имею черного пояса, хотя и надеюсь, что Тейлор Отвелл мне всё-таки его пришлет.

Прогресс не останавливается, PHP развивается уже в течение 25 лет, выпустил уже 10 версий Laravel, всегда появляются какие-то приёмы, сервисы и методы, которые нужно изучать и внедрять в свою практику. Развитие бесконечно и это прекрасно. Добро пожаловать в ряды ninja Laravel-разработчиков! Да благословит нас Код!

И не забывайте, что лучший Laravel гайд - документация и практика! Ну и обучающие видео с youtube-канала CutCode.

ЧТО ДАЛЬШЕ?

Итак, друзья, мы вместе прошлись по функционалу Laravel. Что делать дальше? Этот вопрос я задал себе 6 лет назад, когда начал изучать Laravel и прочитал официальную документацию. Ресурсов на русском языке почти не было, я делал проекты, параллельно изучая статьи и руководства на английском, вооружившись переводчиком. Предлагаю Вам достигнуть цель изучения Laravel дорогой покороче, со мной в качестве проводника.

1. Итак, мой обучающий youtube канал CutCode:

   https://www.youtube.com/c/CutCodeRu

   Более 200 обучающих видео с ответами на самые распространённые вопросы по Laravel.

2. Laravel комьюнити CutCode в Telegram. Обязательно присоединяйтесь! Тут и советом всегда помогут, и новости по Laravel публикуются, и пообщаться можно.

   https://t.me/laravel_chat

3. Панель администратора для проектов на Laravel MoonShine:

   https://moonshine-laravel.com

   Мой open-source проект поможет в администрировании Ваших сайтов. Мощный функционал, классный дизайн, активное комьюнити и подробная документация - Вам обязательно понравиться!

4. Видеокурсы. Отличный способ развиваться! Я выпустил несколько курсов, рассчитанных на разный уровень подготовки. Вы обязательно найдёте что-то интересное и полезное для себя.
   Все курсы доступны на моем сайте - https://learn.cutcode.dev.

ДАВАЙТЕ ПОЗНАКОМИМСЯ. МЕНЯ ЗОВУТ ДАНИЛ ЩУЦКИЙ, И Я ФАНАТ LARAVEL:)
МНЕ 35 ЛЕТ. С ДЕТСТВА ВЫБРАЛ ПРОФЕССИЮ РАЗРАБОТЧИКА И ПОЛУЧИЛ ВЫСШЕЕ ТЕХНИЧЕСКОЕ ОБРАЗОВАНИЕ. УЖЕ 13 ЛЕТ ЗАНИМАЮСЬ WEB-РАЗРАБОТКОЙ, И ПРОШЕЛ ПУТЬ ОТ ВЕРСТКИ ДО ПРОФЕССИОНАЛЬНОГО РАЗРАБОТЧИКА.

В НАСТОЯЩЕЕ ВРЕМЯ КОНСУЛЬТИРУЮ БОЛЬШИЕ КОМАНДЫ РАЗРАБОТЧИКОВ, УЧУ ИХ КАК РАБОТАТЬ С ВЫСОКОНАГРУЖЕННЫМИ ПРОЕКТАМИ, ВЫСТРАИВАТЬ СЛОЖНУЮ АРХИТЕКТУРУ И ВЗАИМОДЕЙСТВИЕ. ЗАПИСЫВАЮ УРОКИ ПО LARAVEL ДЛЯ НОВИЧКОВ И НЕ ТОЛЬКО.

ПОЧТИ КАЖДЫЙ ДЕНЬ РАБОТАЮ С LARAVEL, ЗНАЮ ЕГО ВДОЛЬ И ПОПЕРЕК, НО ВСЁ РАВНО ПРОДОЛЖАЮ УЧИТСЯ, ПОСТОЯННО ИЗУЧАЯ НОВЫЕ ВОЗМОЖНОСТИ ЭТОГО КРУТОГО PHP ФРЕЙМВОРКА. СЧИТАЮ, ЧТО ОБУЧЕНИЕ БЕСКОНЕЧНО, И ЭТО ПРЕКРАСНО!

У МЕНЯ ОГРОМНЫЙ ПРАКТИЧЕСКИЙ ОПЫТ И ЗНАНИЯ ПО РАЗРАБОТКЕ НА LARAVEL. ЧТОБЫ ДОСТИГНУТЬ ЭТОГО УРОВНЯ Я ЗАТРАТИЛ УЙМУ ВРЕМЕНИ! ГОТОВ НАУЧИТЬ И ВАС, ПРИ ЭТОМ ПОМОГУ СЭКОНОМИТЬ И ВРЕМЯ (А ЭТО ДЕНЬГИ) И НЕРВЫ!

МНЕ НРАВИТСЯ ДЕЛИТЬСЯ СВОИМИ ЗНАНИЯМИ, И Я ВИЖУ СВОЮ МИССИЮ В ПОПУЛЯРИЗАЦИИ LARAVEL. РАЗВИВАЮ РУССКОЯЗЫЧНОЕ КОМЬЮНИТИ ПО LARAVEL - CUTCODE. С НАЧАЛА 2021 ГОДА ДЕЛЮСЬ СВОИМ ОПЫТОМ НА YOUTUBE-КАНАЛЕ ПО WEB-РАЗРАБОТКЕ НА LARAVEL, НА КОТОРОМ УЖЕ БОЛЕЕ 10000 ПОДПИСЧИКОВ. МОЖЕТЕ ПОСМОТРЕТЬ МОИ ВИДЕОУРОКИ:
HTTPS://WWW.YOUTUBE.COM/C/CUTCODERU
А ОБЩАЕМСЯ В ТЕЛЕГРАМ:
HTTPS://T.ME/LARAVEL_CHAT