### **Заметки**

#### **1. Глобальная установка Laravel**

Чтобы установить Laravel глобально и использовать его для создания новых проектов, выполните следующую команду:

composer global require laravel/installer

Это добавит Laravel в список глобальных Composer-зависимостей, и его можно будет вызывать напрямую через терминал.

#### **2. Установка нового проекта Laravel**

После установки можно создать новый проект, используя команду:

laravel new example-app

Здесь example-app — имя папки проекта. Laravel автоматически создаст новую структуру папок и установит все необходимые зависимости.

#### **3. Создание контроллера**

Laravel предоставляет команду для быстрого создания контроллеров. Для создания контроллера SiteController выполните:

php artisan make:controller SiteController

Это создаст файл SiteController.php в папке app/Http/Controllers, где будут располагаться методы, связанные с логикой обработки запросов.

#### **4. Установка API в Laravel**

Команда для установки базовой структуры API:

php artisan install:api

(Примечание: убедитесь, что используемая версия Laravel поддерживает эту команду. В последних версиях Laravel API уже встроен, и эта команда может быть неактуальна.)

#### **5. Миграции в Laravel**

Миграции позволяют создавать, изменять и управлять структурой базы данных с помощью кода. Основные команды:

**Запуск миграций**: выполните все доступные миграции (например, для создания таблиц) с помощью:  
php artisan migrate

**Откат последней миграции**: Чтобы отменить последнюю операцию миграции, выполните:  
php artisan migrate:rollback

Эти команды помогают легко контролировать изменения в структуре базы данных в процессе разработки.

#### **6. Создание кастомного запроса**

Кастомные запросы полезны для валидации данных, передаваемых в контроллеры. Для создания нового запроса RegistrationRequest используйте:

php artisan make:request RegistrationRequest

Запрос создается в папке app/Http/Requests и может использоваться для добавления правил валидации данных перед их передачей в контроллер.

#### **7. Пример создания кастомного атрибута**

Laravel позволяет создавать «виртуальные» атрибуты, которые не хранятся в базе данных, но доступны в модели. Пример с использованием метода getFullNameAttribute:

**Метод getFullNameAttribute()**:  
// Создаем виртуальный атрибут "full\_name" для объединения данных

public function getFullNameAttribute()

{

return $this->last\_name . ' ' . $this->first\_name . ' ' . $this->patronymic;

}

**С использованием класса Attribute** (доступно в Laravel 9 и выше):  
public function fullName(): Attribute

{

return Attribute::get(fn () => $this->last\_name . ' ' . $this->first\_name . ' ' . $this->patronymic);

}

Эти методы позволяют получить full\_name в любом месте, где используется экземпляр модели, например, $user->full\_name.

Сосредоточимся на работе с отношениями hasOne и hasMany, политиками (policies) и ресурсами (resources) в Laravel.

#### **8. Работа с отношениями hasOne и hasMany**

Laravel предоставляет удобные методы для работы с отношениями между моделями. Вот основные примеры использования.

**Связь hasOne**:

Отношение «один к одному» (hasOne) используется, когда одна запись в таблице может быть связана с одной записью в другой таблице.

Например, если у нас есть таблица users и таблица profiles, где каждому пользователю соответствует один профиль, мы можем определить отношение hasOne в модели User:  
  
class User extends Model

{

public function profile()

{

return $this->hasOne(Profile::class);

}

}

**Связь hasMany**:

Отношение «один ко многим» (hasMany) используется, когда одна запись в таблице может иметь множество связанных записей в другой таблице.

Например, у каждого пользователя может быть много статей (articles), тогда в модели User мы можем определить hasMany:  
  
class User extends Model

{

public function articles()

{

return $this->hasMany(Article::class);

}

}

#### **9. Создание миграций**

Миграции позволяют определять и изменять структуру базы данных с помощью кода, что облегчает управление изменениями в базе данных.

**Создание новой миграции**:

Команда для создания новой миграции, например, для таблицы gagarin\_flights:  
  
php artisan make:migration create\_gagarin\_flights\_table

Это создаст файл в папке database/migrations, где можно определить структуру таблицы gagarin\_flights.

#### **10. Ресурсы (Resources)**

Ресурсы Laravel позволяют форматировать данные перед отправкой их в API или в клиентское приложение, благодаря чему можно легко контролировать структуру ответа.

**Создание ресурса**:

Чтобы создать ресурс LunarMissionResource, выполните команду:  
  
php artisan make:resource LunarMissionResource

Ресурс создается в папке app/Http/Resources. Он позволяет настроить, какие данные и в каком виде будут возвращаться клиенту.

Пример использования в ресурсе:  
  
class LunarMissionResource extends JsonResource

{

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'launch\_date' => $this->launch\_date,

'status' => $this->status,

];

}

}

#### **11. Политики (Policies)**

Политики Laravel (policies) позволяют контролировать доступ к определенным действиям в приложении, например, разрешить пользователю редактировать или удалять ресурсы.

**Создание политики**:

Политики обычно создаются для моделей. Например, чтобы создать политику для модели Post, выполните команду:  
  
php artisan make:policy PostPolicy

Политика будет создана в папке app/Policies, и в ней можно определить методы для различных разрешений (например, update, delete, view).

Пример метода в политике PostPolicy:  
  
class PostPolicy

{

public function update(User $user, Post $post)

{

return $user->id === $post->user\_id;

}

}

Здесь проверяется, что только автор поста может его обновить.

**Применение политики**:

После создания политики нужно зарегистрировать ее в AuthServiceProvider, чтобы Laravel знал, какую политику применять к модели:  
  
protected $policies = [

Post::class => PostPolicy::class,

];

#### **12. Работа с Postman**

При работе с API можно использовать Postman для отправки запросов и тестирования ответов. Если для авторизации используется токен, его можно сохранить в переменной окружения для последующих запросов:

pm.environment.set("token", pm.response.json().token);

Это позволяет автоматизировать передачу токена в заголовке запроса, делая процесс тестирования API более удобным.

#### **13. Проверка загрузки изображения и добавление водяного знака**

При загрузке изображения в Laravel можно проверять его валидность и добавлять водяной знак.

**Проверка загрузки файла**:

Проверяем, был ли загружен файл и является ли он валидным.  
  
if ($request->hasFile('image') && $request->file('image')->isValid()) {

// Получаем путь к загруженному изображению

$imagePath = $request->file('image')->getPathname();

**Добавление водяного знака на изображение**:

Устанавливаем текст водяного знака, цвет и позицию на изображении.  
// Текст водяного знака

$text = "Watermark Text"; // Замените на нужный текст

return response()->stream(function () use ($imagePath, $text) {

$image = imagecreatefromjpeg($imagePath);

// Определяем цвет текста (например, белый)

$textColor = imagecolorallocate($image, 255, 255, 255);

// Устанавливаем параметры шрифта

$fontSize = 5; // Размер шрифта (1-5 для встроенных шрифтов PHP)

$margin = 10; // Отступ от края изображения

// Получаем ширину и высоту текста

$textWidth = imagefontwidth($fontSize) \* strlen($text);

$textHeight = imagefontheight($fontSize);

// Рассчитываем позицию текста в правом нижнем углу

$x = imagesx($image) - $textWidth - $margin;

$y = imagesy($image) - $textHeight - $margin;

// Добавляем текст на изображение

imagestring($image, $fontSize, $x, $y, $text, $textColor);

// Устанавливаем заголовок для вывода изображения

header('Content-Type: image/jpeg');

// Отправляем изображение пользователю

imagejpeg($image);

// Освобождаем память

imagedestroy($image);

}, 200, ['Content-Type' => 'image/jpeg']);

Если изображение недействительно, возвращаем ошибку:  
  
return response()->json(['error' => 'Invalid image upload'], 400);

#### **14. Использование регулярных выражений (regex) для проверки данных**

**Проверка, что строка начинается с заглавной буквы** (поддерживает кириллицу и латиницу):  
  
regex:/^[А-ЯA-Z]/u

**Проверка пароля**:

Должен содержать хотя бы одну строчную букву, одну заглавную букву и одну цифру, минимум 3 символа.  
  
regex:/^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d).{3,}$/u

**Публикация конфигурации CORS**

Чтобы изменить настройки CORS (Cross-Origin Resource Sharing), можно опубликовать конфигурацию:

php artisan config:publish cors

Это позволяет настроить политику доступа для API и других ресурсов.

#### **Создание правила валидации**

В Laravel можно создать кастомные правила валидации для проверки данных.

**Создание правила**:

Команда для создания правила FirstLetterRule, которое проверяет первую букву строки:  
  
php artisan make:rule FirstLetterRule

**Пример реализации правила**:

После создания правила его можно использовать в валидаторах для проверки данных, например, чтобы строка начиналась с определенной буквы.